Árpád Gellért Lucian N. Vinţan Adrian Florea

A Systematic Approach to Predict Unbiased Branches

"Lucian Blaga" University Press
Sibiu 2007

Tiparul executat la:
Compartimentul de Multiplicare al
Editurii Universității „Lucian Blaga“ din Sibiu,
B-dul Victoriei nr. 10, Sibiu 550024
Tel.: 0269 210 122
E-mail: editura@ulbsibiu.ro
claudiu.fulea@ulbsibiu.ro

Descrierea CIP a Bibliotecii Naționale a României

GELLÉRT, ÁRPÁD
A systematic approach to predict unbiased branches /
Gellért Árpád, Vințan N. Lucian, Florea Adrian. - Sibiu : Editura Universității "Lucian Blaga" din Sibiu, 2007
Bibliogr.
Index
ISBN 978-973-739-516-0

I. Vințan, Lucian N.
II. Florea, Adrian

004

Acknowledgments

This work was supported by the Romanian Agency for Academic Research (CNCSIS) through our research grants TD-248/2007-2008 respectively 39/2007-2008. It was also partially carried out under the HPC-EUROPA project (RII3-CT-2003-506079), with the support of the European Community – Research Infrastructure Action under the FP6 "Structuring the European Research Area" Programme.

We express our gratitude to Professor Theo UNGERER, PhD, from the University of Augsburg, Germany, for the useful discussions and for all his various support. Also our gratitude to Dr. Colin EGAN from the University of Hertfordshire, UK, for his research collaboration for over 10 years. Our full recognition to our MSc students Ciprian RADU, Horia CALBOREAN and Adrian CRAPCIU, from "Lucian Blaga" University of Sibiu, who were actively involved in implementing the simulator presented in Chapter 6, to our colleague Marius OANCEA, who has also contributed in the unbiased branch research, his results being presented in paragraph 3.2.1, respectively to Mihai MUNTENAŞ, MSc, who brought contributions in paragraphs 3.2.2 and 4.6.1.

The authors

Contents

1. Introduction into Unbiased Branches Challenge 7

2. Related Work 9

3. Finding Difficult-to-Predict Branches 24

3.1. Methodology of Identifying Unbiased Branches 24

3.2. Experimental Results 27

3.2.1. Pattern-based Correlation 28

3.2.2. Path-based Correlation 39

3.2.3. An Analytical Model 45

3.2.4. An Example Regarding Branch Prediction Contexts Influence 49

4. Predicting Unbiased Branches 54

4.1. The Perceptron-Based Branch Predictor 54

4.2. The Idealized Piecewise Linear Branch Predictor 55

4.3. The Frankenpredictor 57

4.4. The O-GEHL Predictor 57

4.5. Value-History-Based Branch Prediction with Markov Models 58

4.5.1. Local Branch Difference Predictor 60

4.5.2. Combined Global and Local Branch Difference Predictor 61

4.5.3. Branch Difference Prediction by Combining Multiple Partial Matches 62

4.6. Experimental Results 64

4.6.1. Evaluating Neural-Based Branch Predictors 64

4.6.2. Evaluating the O-GEHL Predictor 69

4.6.3. Evaluating Local Branch Difference Predictors 69

4.6.4. Evaluating Combined Global and Local Branch Difference Predictors 75

4.6.5. Branch Difference Prediction by Combining Multiple Partial Matches 78

5. Using Last Branch Difference as Prediction Information 81

6. Designing an Advanced Simulator for Unbiased Branches Prediction 88

6.1. Simulation Methodology **89**
6.2. The Functional Kernel of the Simulator **90**
6.3. The Software Design of the ABPS Simulator **91**
7. Conclusions and Further Work ***98***
References ***101***
Glossary ***108***

1. Introduction into Unbiased Branches Challenge

Two trends – technological and architectural (conceptual) – are further increasing the importance of branch prediction. From technological point of view, modern high-end processors use an array of tables for branch direction and target prediction [Sez02]. These tables are quite large in size (352K bits for the direction predictor in Alpha EV8) and they are accessed every cycle resulting in significant energy consumption – sometimes more than 10% of the total chip power [Cha03].

From an architectural point of view, processors are getting wider and pipelines are getting deeper, allowing more aggressive clock rates in order to improve overall performance. A very high frequency will determine a very short clock cycle and the prediction cannot be delivered in a single clock cycle or maximum two cycles which is the prediction latency in the actual commercial processors (see Alpha 21264 branch predictor) [Jim02]. Also a very wide superscalar processor can suffer from performance point of view in the misprediction case when the CPU context must be recovered and the correct paths have to be (re)issued. As an example, the performance of the Pentium 4 equivalent processor degrades by 0.45% per additional misprediction cycle, and therefore the overall performance is very sensitive to branch prediction. Taking into account that the average number of instructions executed per cycle (IPC) grows non-linearly with the prediction accuracy [Yeh92], it is very important to further increase the accuracy achieved by present-day branch predictors.

The quality of a prediction model is highly dependent on the quality of the available data. Especially the choice of the *features* to base the prediction on is important. The vast majority of branch prediction approaches rely on usage of a greater number of input features (such as branch address, global or local branch history, etc.) without taking into account the real cause (unbiased branches) that produce a lower accuracy and implicit lower performance.

In this work we prove that a branch in a certain dynamic context is difficult predictable if it is unbiased and the outcomes are shuffled. In other words, a dynamic branch instruction is unpredictable with a given prediction information if it is unbiased in the considered dynamic context and the

behavior in that certain context cannot be modeled through Markov stochastic processes of any order. Based on laborious simulations we show that the percentages of difficult branches are quite significant (at average between 6% and 24%, depending on the different used prediction contexts and their lengths), giving a new research challenge and a useful niche for further research. Present-day branch predictors are using limited prediction information (local and global correlation and path information). We'll show that for some branches this information is not always sufficiently relevant and, therefore, these branches cannot be accurately predicted using present-day predictors. Consequently, we think it is important to find other relevant information that is determining branches' behavior in order to use it for designing better predictors. In our opinion such relevant prediction information could consist in branch's condition sign (positive, negative or zero). More precisely, a certain branch associated with its condition's sign value (+, -, 0) will be perfectly biased. If its condition sign will be predictable, the branch's behavior will be predictable too because the branch's output is deterministically correlated with the condition's sign. Thus, it appears rationale trying to predict current branch's condition sign based on the local/global condition histories. We can also use the last branch condition as new prediction information in some state-of-the-art branch predictors in order to increase prediction accuracy.

This booklet is organized as follows. Chapter 2 gives a brief overview of related work. Chapter 3 describes our methodology of finding difficult predictable branches. Chapter 4 describes the present-day branch predictors used in this work and continues with some proposed condition-history-based branch prediction methods. Chapter 5 presents some modified present-day branch predictors that use the last known branch condition as prediction information. Chapter 6 presents an advanced simulator for unbiased branches' prediction. Finally, Chapter 7 concludes the booklet and suggests directions for further work.

2. Related Work

Representative hardware and compiler-based branch prediction methods have been developed in recent years in order to increase instruction-level parallelism. Branch prediction is an important component of modern microarchitectures, despite of their deeper pipelines that increased misprediction latency. Therefore, improvements in terms of branch prediction accuracy are essential in order to avoid the penalties of mispredictions. In this section we presented only the works that are most closely related to the proposed approach.

Chang et al., introduced in [Cha94] a mechanism called branch classification in order to enhance branch prediction accuracy by classifying branches into groups of highly biased (mostly-one-direction branches) respectively unbiased branches, and used this information to reduce the conflict between branches with different classifications. In other words, they proposed a method that classifies branches according to their dynamic taken rate and assigns branches from each class to different predictors. The class of branches is determined by their overall dynamic taken rate collected during program profiling. With their branch classification model they showed that using a short history for the biased branches and a long history for the unbiased branches improves the performance of the global history Two-Level Adaptive Branch predictors. In contrast to our work, the authors are classifying branches irrespective of their attached context (local and global histories, etc.) involving thus an inefficient approach. Due to this rough classification the corresponding predictors are not optimally chosen, simply because it is impossible to find an optimal predictor for some classes.

Mahlke et al., proposed in [Mah94] a compiler technique that uses predicated execution support to eliminate branches from an instruction stream. Predicated execution refers to the conditional execution of an instruction based on the value of a boolean source operand – the predicate of the instruction. This architectural support allows the compiler to convert conditional branches into predicate defining instructions, and instructions along alternative paths of each branch into predicated instructions. Predicated instructions are fetched regardless of their predicate value. Thus, instructions whose predicate value is true are executed normally, while instructions whose predicate is false are nullified. Predicated execution

offers the opportunity to improve branch handling in superscalar processors. Eliminating frequently mispredicted branches may lead to a substantial reduction in branch prediction misses, and as a result, the performance penalties associated with the eliminated branches are removed. The authors use compiler support for predicated execution based on a structure called hyperblock. The goal of hyperblock formation is to group basic blocks eliminating unbiased branches and leaving highly biased branches. They selected the unbiased branches based on taken frequency distributions. Their experimental results show that leaving only highly biased branches with predicated execution support, the prediction accuracy is higher.

Nair has first introduced dynamic branch prediction based on path correlation [Nair95]. The basic observation behind both pattern-based and path-based correlation is that some branches can be more accurately predicted if the path leading to these branches is known. Path-based correlation attempts to overcome the performance limitations of pattern-based correlation arising from pattern aliasing situations, where knowledge of the path leading to a branch results in higher predictability than knowledge of the pattern of branch outcomes along the path. Nair proposed a hardware scheme which records the path leading to a conditional branch in order to predict the outcome of the branch instruction more accurately. He adapted a pattern-based correlation scheme, replacing the pattern history register with a g-bit path history register which encodes the target addresses of the immediately preceding p conditional branches. Ideally, all bits of the target address should be used to ensure that each sequence of p addresses has a unique representation in the register. Since such schemes are too expansive to be implemented in hardware, Nair used a simplified scheme which uses a subset of q bits from each of the target addresses. Limiting the number of bits from the branch address could result path aliasing – the inability of the predictor to distinguish two distinct paths leading to a branch. Unfortunately, this path correlation scheme does not show any significant improvement over pattern-based correlation [Nair95]. Nair's explanation for this is that for a fixed amount of hardware in the prediction tables, path-based correlation uses a smaller history than pattern-based correlation because the same number of bits represents fewer basic blocks in the path history register than branch outcomes in the pattern history register. Despite this, path based correlation is better than pattern-based correlation on some benchmarks – especially when history information is periodically destroyed due to context switches –, indicating that with a better hashing scheme the pattern correlation schemes could be outperformed.

A quite similar approach is proposed by Vintan and Egan in [Vin99b] – their paper represents the genesis of this work. The authors

illustrated, based on examples, how a longer history could influence the behavior of a branch (changing it from unbiased to biased). They also showed that path information could also reduce the branch's entropy. The main contribution of this paper is related to the prediction accuracy gain obtained by extending the correlation information available in the instruction fetch stage. Based on trace-driven simulation the authors proved for relatively short global branch history patterns, that a path-based predictor overcomes a pattern-based predictor at the same hardware budget. The main difference, comparing with Nair's approach, is that here the authors are using both the path and respectively the history information in order to do better predictions. They show that a scheme based on this principle performs better than a classical GAp scheme, at the same level of complexity. Particularly useful information has been gleaned regarding the interaction between path length and the number of replacements required in the PHT.

Dynamic branch prediction with neural methods, was first introduced by Vintan [Vin99a, Ega03], and further developed by Jiménez [Jim01]. Despite the neural branch predictor's ability to achieve very high prediction rates and to exploit deep correlations at linear costs, the associated complexity due to latency, large quantity of adder circuits, area and power are still obstacles to the industrial adoption of this technique. Anyway, the neural methods seem to be successfully for future microprocessors taking into account that they are already implemented in Intel's IA-64 simulators. The path-based neural predictors [Jim03] improve the instructions-per-cycle (IPC) rate of an aggressively clocked microarchitecture by 16% over the original perceptron predictor [Jim01]. A branch may be linearly inseparable as a whole, but it may be piecewise linearly separable with respect to the distinct associated program paths. More precisely, the path-based neural predictor combines path history with pattern history, resulting superior learning skills to those of a neural predictor that relies only on pattern history. The prediction latency of path-based neural predictors is lower, because the computation of the output can begin in advance of the prediction, each step being processed as soon as a new element of the path is executed. Thus, the vector of weights used to generate prediction, is selected according to the path leading up to a branch – based on all branch addresses from that path – rather than according to the current branch address alone as the original perceptron does. This selection mechanism improves significantly the prediction accuracy, because, due to the path information used in the prediction process, the predictor is able to exploit the correlation between the output of the branch being predicted and the path leading up to that branch. To generate a prediction, the correlations

of each component of the path are aggregated. This aggregation is a linear function of the correlations for that path. Since many paths are leading to a branch, there are many different linear functions for that branch, and they form a piecewise-linear surface separating paths that lead to predicted taken branches from paths that lead to predicted not taken branches. The piecewise linear branch prediction [Jim05], is a generalization of neural branch prediction [Jim01], which uses a single linear function for a given branch, and respectively path-based neural branch prediction [Jim03], which uses a single global piecewise-linear function to predict all branches. The piecewise linear branch predictors use a piecewise-linear function for a given branch, exploiting in this way different paths that lead to the same branch in order to predict otherwise linearly inseparable branches. The piecewise linear branch predictors exploit better the correlation between branch outcomes and paths, yielding an IPC improvement of 4% over the path-based neural predictor [Jim05].

A conventional path-based neural predictor achieves high prediction accuracy, but its very deeply pipelined implementation makes it both a complex and power-intensive component, since for a history length of *p* it uses – to store the weights – *p* separately indexed SRAM arrays organized in a p-stage predictor pipeline. Each pipeline stage requires a separate row-decoder for the corresponding SRAM array, inter-stage latches, control logic and checkpointing support, all of this adding power and complexity to the predictor. Loh and Jiménez proposed in [Loh05c] two techniques to address this problem. The first decouples the branch outcome history length from the path history length using shorter path history and a traditional long branch outcome history. In the original path-based neural predictor, the path history was always equal to the branch history length. The shorter path history allows the reduction of the pipeline length, resulting in decreased power consumption and implementation complexity. The second technique uses the bias-weights to filter out highly-biased branches (mostly always taken or mostly always not taken branches), and avoids consuming update power for these easy-to-predict branches. For these branches the prediction is determined only by the bias weight, and if it turns out to be correct, the predictor skips the update phase which saves the associated power. The proposed techniques improve the prediction accuracy with 1%, and more important, reduce power and complexity by decreasing the number of SRAM arrays, and reducing predictor update activity by 4-5%. Decreasing the pipeline depth to only 4-6 stages it is reduced the implementation complexity of the path-based neural predictor.

Tarjan and Skadron introduced in [Tar05] the hashed perceptron predictor, which merges the concepts behind the gshare [McFar93] and

path-based perceptron predictors [Jim03]. The previous perceptron predictors assign one weight per local, global or path branch history bit. This means that the amount of storage and the number of adders increases linearly with the number of history bits used to make a prediction. One of the key insights of Tarjan's work is that one-to-one ratio between weights and number of history bits is not necessary. By assigning a weight not to a single branch but a sequence of branches (hashed indexing), a perceptron can work on multiple partial patterns making up the overall history. The hashed indexing consists in XORing a segment of the global branch history with a branch address from the path history. Decoupling the number of weights from the number of history bits used to generate a prediction allows the reduction of adders and tables almost arbitrarily. Using hashed indexing, linearly inseparable branches which are mapped to the same weight can be accurately predicted, because each table acts like a small gshare predictor [McFar93]. The hashed perceptron predictor improves accuracy by up to 27.2% over a path-based neural predictor.

Loh and Jiménez introduced in [Loh05b] a new branch predictor that takes the advantage of deep-history branch correlations. To maintain simplicity, they limited the predictor to use conventional tables of saturating counters. Thus, the proposed predictor achieves neural-class prediction rates and IPC performance using only simple PHT (pattern history table) structures. The disadvantage of PHTs is that their resource requirements increase exponentially with branch history length (a history length of p requires 2^p entries in a conventional PHT), in contrast to neural predictors, whose size requirements increase only linearly with the history length. To deal with very long history lengths, they proposed a Divide-and-Conquer approach where the long global branch history register is partitioned into smaller segments, each of them providing a short branch history input to a small PHT. A final table-based predictor combines all of these per-segment predictions to generate the overall decision. Their predictor achieves higher performance (IPC) than the original global history perceptron predictor, outperforms the path-based neural predictors, and even achieves an IPC rate equal to the piecewise-linear neural branch predictor. Using only simple tables of saturating counters, it is avoided the need for large number of adders, and in this way, the predictor is feasible to be implemented in hardware.

Desmet et al. [Des04] proposed a different approach for branch classification. They evaluated the predictive power of different branch prediction features using *Gini-index* metric, which is used as selection measure in the construction of decision trees. Actually, *Gini-index* is a metric of informational energy and in this case is used to identify the

branches with high entropy. In contrast to our work Desmet used as input features both dynamic information (global and local branch history) and static information (branch type, target direction, ending type of taken-successor-basic-block).

In [Hei99a] the authors identified some program constructs and data structures that create “hard to predict” branches. In order to accurately predict difficult branches the authors find additional correlation information beyond local and global branch history. In their approach the prediction table is addressed by a combination between structural information, value information and history of values that are tested in the condition of respective branch. Unlike our work, Heil et al. didn’t use the path history information in order to do better predictions. Using the proposed prediction method based on data values significantly improves prediction accuracy for some certain difficult branches but the overall improvements are quite modest. However there are some unsolved problems: they tested only particular cases of difficult branches, and also, they didn’t approach branch conditions with two input values. Their final conclusion suggests that researchers must focus on the strong correlation between instructions producing a value and, respectively, the branch condition that would be triggered by that certain value.

In [Cha02] the authors are focusing on some difficult predictable branches in a Simultaneous Subordinate Micro-Threading (SSMT) architecture. They defined a difficult path being a path that has a terminating branch which is poorly predicted when it executes from that path. A path represents a particular sequence of control-flow changes. It is shown that between 70% and 93.5% of branch mispredictions are covered by these difficult paths, involving thus a significant challenge in branch prediction paradigm. The proposed solution in dealing with these difficult predictable branches consists in dynamically construct micro-threads that can speculatively and accurately pre-compute branch outcomes, only along frequently mispredicted paths. Obviously, micro-thread predictions must arrive in time to be useful. Ideally, every micro-thread would complete before the fetch of the corresponding difficult branch. By observing the data-flow within the set of instructions guaranteed to execute each time the path is encountered, it can be extracted a subset of instructions that will pre-compute the branch. The proposed micro-architecture contains structures to dynamically identify difficult paths (Path Cache), construct micro-threads (Micro-Thread Builder) and communicate predictions to the main thread. The proposed technique involves realistic average speed-ups of up to 10% but the average potential speed-up through perfect prediction of these difficult branches is about 100%, suggesting the idea’s fertility.

Unfortunately the authors didn't investigate why these paths, respectively their associated final branches, are difficult predictable. In other words, a very important question is: why these "difficult paths" frequently lead to miss-predictions? We could suspect that we already gave the answer in our paper because these "difficult branches" might be, at least partially, exactly the unbiased branches in the sense defined by us, and, therefore, difficult predictable. They could be more predictable even in a single threaded environment, by sufficiently growing history pattern length or extending prediction information, as we'll show further in this work. Thus, our hypothesis is that SSMT environment represents a sufficient solution in order to solve these difficult branches, as the authors shown, but not a necessary one.

In [Che03] the authors proposed a new approach, called ARVI (Available Register Value Information), in order to predict branches based on partial register values along the data dependence chain leading up to the branch. The authors show that for some branches the correlation between such register value information and the branch's outcome can be stronger than either history or path information. Thus, the main idea behind the ARVI predictor is the following: if the essential values in the data dependence chain, that determine the branch's condition, should be identified, and those values have occurred in the past, then the branch's outcome should be known. If the values involved in the branch condition are the same as in a prior occurrence then the outcome of the branch will be the same, too. Thus, if the branch's register values are available then a look up table can provide the last branch's outcome occurred with the same values. Unfortunately, the branch's register values are rarely available at the time of prediction. However, if values are available for registers along the dependence chain that leads up to the branch, then the predictor can use these values to index into a table and reuse the last behavior of the branch occurred in the same context. Therefore, instead of relying only on branch history or path, the ARVI predictor includes the data dependent registers as part of the prediction information. The ARVI predictor uses a Data Dependence Table (DDT) to extract the registers corresponding to instructions along the data dependence chain leading up to the branch. The branch's PC and the identifiers of the data dependent registers are hashed together and used to index the prediction table. The values of the data dependent registers are hashed together and used as a tag to distinguish the occurrences of the same path having different values in the registers. Thus, the ARVI predictor uses both path and value-based information to classify branch instances. A two-level predictor using ARVI at the second level achieves a 12.6% overall IPC improvement over the state-of-the-art two

level predictors, for the SPEC'95 integer benchmarks. The authors selected SPEC'95 integer benchmarks because their branch behavior was extensively studied permitting comparisons between different works. In our opinion, if dynamic branches that are unbiased in their branch history or path contexts [Vin06] are biased in their value history context, the benefit could be remarkable. An analysis in this sense should be effectuated.

Z. Smith in his work [Smi98] determined through simulation on the SPEC'95 benchmarks that the majority of branch mispredictions come from a relatively small number of static branches. Therefore, he identified "bad" branches based on the distribution of mispredictions – a function of the number of mispredictions per branch using the *gshare* predictor with 12 history bits. An analysis of branches having a relatively high number of mispredictions shows that they could be really less predictable but without importance due to their relatively low number of dynamic instances, and, on the other hand, some of them could be predictable because the number of mispredictions is, however, far less then the number of branch's dynamic instances. Consequently, there is no strong correlation between branch's predictability or global prediction accuracy and the distribution of mispredictions. In order to increase the predictability of mostly mispredicted branches, Smith evaluated the possibility to predict branch outcomes based on a value history. The idea is to use a context-based predictor whose prediction table is indexed by a register value instead of the XOR between the PC and global history as in *gshare*. In their implementation, only the first (non-immediate) branch operand is used as prediction context, because, as he shows, the majority of branches have the second operand equal with zero. However, using both branch operands as prediction information could be better. Using a history of only 2 values together with the value of the outer loop counter (an iteration counter associated to the enclosing loop's branch), Smith obtained a branch prediction accuracy of 93.4%.

In [Hei99b] the authors observed that many important branches that are hard to predict based on branch history and path become easily predictable if data-value information is used. First, they analyzed a technique called *speculative branch execution* that uses a conventional data-value predictor to predict the input values of the branch instruction and, after that, executes the branch instruction using the predicted values. The main disadvantage of this method consists in the relatively high prediction latency, because the operand-value prediction is followed by the pre-calculation of the branch's condition. Therefore, they proposed a Branch Difference Predictor (BDP) that maintains a history of differences between branch source register operands and uses it in the prediction process. Consequently, the value history information is used directly for branch

prediction, reducing thus the latency. Since branch outcomes are determined by subtracting the two inputs, the branch source differences correlate very well with the branch outcomes. The branch difference history is maintained per static branch in a Value History Table (VHT) and it is retrieved using the branch's PC. By using branch differences, the number of patterns is very high, since a certain static branch instruction may produce many values. Thus, predicting all branches through this method leads either to excessive storage space requirements or to significant table interference. Therefore, in their prediction mechanism, only the difficult branches are predicted based on the branch source differences using the Rare Event Predictor (REP), while most branches are predicted using a conventional predictor (e.g. *gshare*). They considered that a branch is difficult if it is mispredicted by the conventional predictor. Therefore, REP's updating introduces only branches mispredicted by the conventional predictor but correctly predicted by REP. When a branch instruction occurs, the VHT and the REP are accessed in parallel with the PC and global branch history. If the value difference history matches a REP tag, then the REP provides the prediction. If the REP does not contain that certain pattern, the conventional branch predictor generates the prediction. Their results show that the majority of prediction accuracy improvement is gained by using a single branch difference, while adding a second or third difference results in little additional improvement. The BDP reduces the misprediction rate by up to 33% compared to *gshare* and up to 15% compared to *Bi-Mode* predictors, in the SPEC'95 integer benchmarks. A first important difference between Heil's approach and ours is that we are focalizing on unbiased branches identified in our previous work [Vin06] instead of Heil's difficult branches. However, the main difference is that we correlate branch's outcome with the sign of the condition's difference while Heil et al. correlate it with the value of the condition's difference. As we'll further show, using signs instead values involves better prediction accuracies and less storage necessities. Furthermore, we use a sign-history of up to 256 condition differences in contrast to the value-history of up to 3 condition differences exploited in [Hei99b]. Another important difference between the two approaches is the architectural one, since we predict branches using some state-of-the-art Markov and neural predictors.

Thomas et al. [Tho03] introduced new branch prediction information that consists in *affector branches*. They identify for each dynamic branch from a long global history, a set of branches called *affectors*, which control the computation that directly affect the source operands of the current dynamic branch. Since affectors have a direct effect on the outcome of a future branch, they have a high correlation with that branch. The affector

information is represented as a bitmap having all bits corresponding to the affector branches set to 1 and, respectively, those of non-affectors set to 0. The affector information is maintained based on runtime dataflow information for each architectural register as entries in an Affector Register File (ARF). When the processor encounters a conditional branch, all entries in the ARF are shifted left by one bit and the least significant bit is made 0. When a register-writing instruction occurs, the ARF entries corresponding to the source registers are ORed together and written into the ARF entry of the destination register with the least significant bit set to 1. Thus, the affector information for the destination register is generated as a union of the affector histories corresponding to the source registers, while the least significant bit, set to 1, marks the last branch from the global history as an affector. The affector branch information for a branch instruction is inherited from the affector information corresponding to its source registers. Therefore, when a prediction is to be made for a certain branch, the affector information of its source registers are ORed together in order to determine its affector branches. The authors also proposed different prediction schemes that use the affector branch information.

In another work Thomas et al. [Tho01] improved instruction centric value prediction by using a *dynamic dataflow inherited speculative context* (DDISC) for hard-to-predict instructions. The DDISC consists in a compression of the PCs and the predicted values of the predictable source producer instructions. The context is determined by assigning a signature to each node in the dataflow graph. The signature of a predictable instruction is its value predicted by a conventional predictor. The signature of unpredictable non-load instructions is inherited from the signatures of its operand producers. In the case of multiple operands, the signature of unpredictable non-load instructions is the XOR of the signatures of their operand producers. The signature of unpredictable load instructions is inherited from the signature of the preceding store instruction that wrote the value into the same memory location. The DDISC for a certain instruction is obtained by rotating its calculated signature by a value determined by the PC (e.g. the last five bits of the PC). Their simulation results show that introducing dataflow-based contexts the prediction accuracy improvement ranges from 35% to 99%.

Constantinides et al. [Con04] presented a method of detecting instruction-isomorphism and its application to dynamic branch prediction. A dynamic instruction is considered isomorphic if its component graph is identical with the component graph of an earlier executed dynamic instruction. The component graph of a dynamic instruction can include information about the instruction, its dynamic data dependence graph and its

input data. Two cases of instruction isomorphism can be distinguished: isomorphic-equality and pseudo-isomorphism. In the case of isomorphic equality the instructions are isomorphic and they have the same outputs, while in the pseudo-isomorphism case, the instructions are isomorphic but their outputs are not equal. The isomorphism detection process is preceded by component-graph transformations that may convert non-isomorphism to isomorphic-equality by removing information from the component graph that does not affect the outcome of the instruction. The isomorphism detection mechanism contains four units: the Register-Signature File (RSF), the Component Graph Encoding/Transformation mechanism (CGET), the Memory Signature File (MSF) and the Isomorphism Detection Table (IDT). The RSF is accessed with the source register names to read the signatures – encoded component graphs. The CGET takes the instruction's source signatures and creates a new signature, which represents the instruction's encoded/transformed component-graph. If the instruction writes to a register the new signature is written into the RSF entry corresponding to the destination register. To determine if an instruction is isomorphic with a previously executed instruction, its signature – produced by CGET – is used to access the IDT. The IDT also returns the branch direction in the case of branch prediction. Isomorphism detection must wait for decoded instruction information and, thus, the isomorphic branch predictor has relatively high latency. Therefore, Constantinides et al. proposed a hybrid branch prediction mechanism composed by a fast conventional predictor and a slower isomorphic-based predictor. Consequently, the isomorphic prediction – available few cycles after the conventional prediction – is used to validate and possibly override the prediction provided by the fast base predictor.

In [Gon99] and [Gon01] González et al. introduced a *branch prediction through value prediction* unit (BPVP) that pre-computes the outcomes of branches by predicting their input values. Since, the accuracy of value predictors is lower than that of the conventional branch predictors, speculative branch pre-computation must be applied selectively. Therefore, they proposed a hybrid branch prediction mechanism involving a correlating branch predictor (e.g. *gshare*) and a BPVP that uses a conventional value predictor. The value predictor is used together with an Input Information Table (IIT) and, respectively, an additional logic to detect the instructions that generate the branch's inputs. Each architectural register has an entry in the IIT that stores the PC of the latest instruction having the corresponding register as destination and, respectively, the value computed speculatively by the latest *compare* instruction having the corresponding register as destination. The *compare* instructions are speculatively pre-executed according to their predicted inputs and the speculative results are stored in

the IIT. The mechanism has different behaviors depending on the branch that is predicted. In the case of branches with inputs produced by arithmetic or load instructions, the IIT is accessed with the source register names to read the PCs of the latest instructions that had as destination the branch's source registers (detection of the instructions that produces the branch inputs). The PCs are used to access the value predictor that predicts the inputs of the branch. The branch's outcome is speculatively pre-computed based on the predicted inputs. In the case of branches with inputs produced by *compare* instructions, the IIT is accessed with the source register names to read the comparison's speculative result. The outcome of the branch is speculatively pre-computed based on this speculative comparison result. The BPVP-*gshare* predictor achieves a speedup of 8% over the 2bit-gshare predictor. The instruction centric value prediction within the BPVP should be replaced with register centric value prediction [Vin05], reducing the complexity, hardware costs and power consumption. Thus, branches should be pre-computed speculatively based on their input values predicted with an optimized register centric value predictor (2-level adaptive value predictor instead of PPM).

In [Rot99] call targets are correlated with the instructions that produce them rather than with the call's global history or the previous branches' targets. The proposed approach pre-computes virtual function call's (v-call) targets. V-calls' targets are hard predictable even through path-history based schemes that exploit the correlation between multiple v-calls to the same object reference. Object oriented programming increases the importance of v-calls. The proposed technique dynamically identifies the sequence of instructions that computes a v-call target. Based on this instruction sequence it is possible to pre-calculate the target before the actual v-call is encountered. This pre-calculation can be used to supply a prediction. The approach reduces v-call target miss-predictions with 24% over a path-based two level predictor.

In [Vin03] the authors proposed to pre-compute branches instead of predicting them. Pre-computing branches means to determine the outcome of a branch as soon as all branch operands are known. The instruction that produced the last operand also triggers the branch condition estimation and, after this operation, it correspondingly computes the branch outcome. Similarly to branch history prediction, branch information is cached into a "prediction table" (PT). Each PT entry has the following fields: TAG (the lower part of the PC), PC1 and PC2 (the PCs of the instructions that produced the branch operand values), OPC (the opcode of the branch), nOP1 and nOP2 (the register names of the branch operands), PRED (for the branch outcome) and a LRU field (Least Recently Used). The register file

has two additional fields for each register: LP (the PC of the last producer) and RC (a reference counter which is incremented by each instruction that modifies a register linked by a branch instruction stored in the PT and, respectively, decremented when the corresponding branch instruction is evicted from the PT). The PC of any non-branch instruction that modifies at least one register is recorded into the supplementary LP (Last Producer) field of its destination register. The first issue of a particular branch in the program is predicted with a default value (not taken). After the branch's execution, a PT entry is allocated and updated. Every time after a non-branch instruction – having the corresponding RC field greater than 0 – is executed, the PC1 and PC2 fields from the PT are searched upon its PC. When a hit occurs, the branch stored in that PT entry is executed and the outcome is stored into the PRED bit. When the branch is issued, its outcome is found in the PT, as it was previously computed, and thus its behavior is perfectly known before execution. From the pure prediction accuracy point of view this method seems to be almost perfect. Unfortunately, the improvement in prediction accuracy brought by this scheme must be paid in terms of timing – because branches frequently follow too closely after the source producer instructions – and hardware costs. Based on the pre-computing branch concept [Vin03] Aamer et al. presented in [Aam03] a study regarding the number of instructions occurred between the execution of the instruction that produced the last operand of a branch and the execution of that branch. Their simulations show that the average distance between the last source producer and branch is less than the ideal theoretical distance. If the operand producer instruction is too close to the corresponding branch then the branch would have to postpone processing for a few cycles, until the operand producer instruction is finished. For these branches a BTB can be used, improving thus the performance. Thus, the branch outcomes can be obtained far enough in advance so that some performance improvement can be still achieved.

Aragón et al. presented in [Ara01] a new approach to improve branch predictors: *selective branch prediction reversal*. The main idea is that many branch mispredictions can be avoided if they are selectively reversed. Therefore, they proposed a Branch Prediction Reversal Unit (BPRU) that reverses predictions of branches likely to be mispredicted, based on the path leading to the branch (including the PC of the input producers) and, respectively, the predicted values of the branch inputs. The BPRU uses the previously presented BPVP-*gshare* hybrid branch predictor [Gon99] and a Reversal Table (RT). Each entry of the RT stores a reversal counter implemented as an up/down saturating counter, and a tag. When a branch is predicted, the RT is accessed by hashing together the PCs of its

input producers, the predicted input values and the path leading to the branch. The most significant bit of the counter indicates if the predicted branch outcome must be reversed. When the correct branch outcome is available, the corresponding RT entry is updated by incrementing the reversal counter if the preliminary branch outcome was correct and, respectively, decrementing the counter otherwise. The experimental results show average speedups of 6% over the original BPVP-*gshare* and, respectively, of 14% over the 2bit-gshare predictor.

In [Gao06] the authors initially implemented a PPM-based branch predictor using as context the global branch history. They associated a signed saturating prediction counter ranging between [-4, 4] to each PC-history pair. The counter was incremented if the branch outcome was taken and decremented otherwise. When both the branch address and history pattern were matched, the corresponding counter provided the prediction. In the case of multiple matches for a branch with different history lengths, the prediction counter afferent to the longest history was used. However, as they show, the longest history match may not be the best choice, and, therefore, they proposed another scheme called PPM with the confident longest match that uses the prediction counter as a confidence measure. This scheme generates a prediction only when the counter is a non-zero value. The authors observed that in the case of multiple matches with different history lengths, the counters may not agree with each other and different branches may favor different history lengths. Thus, the most important scheme introduced by Gao and Zhou in this paper, predicts branch outcomes by combining multiple partial matches through an adder tree. The *Prediction by combining Multiple Partial Matches* (PMPM) algorithm selects up to *L* confident longest matches and sums the corresponding counters to make a prediction. For the fully biased (always taken or always not taken) branches they use a bimodal predictor, the PMPM predictor being accessed only for not fully biased branches. The realistic PMPM predictor has seven global prediction tables indexed by the branch address, global history and path, and, respectively, a local prediction table indexed by the branch address and local history. When the PMPM is accessed for prediction, up to 4 counters from the global history tables are summed with the counter from the local prediction table, if there is a hit. If the sum is zero, the bimodal predictor is used. Otherwise the sign of the sum provides the prediction. The prediction counter from the bimodal prediction table is always updated. The prediction counter from the local prediction table is always updated in the case of hit, while the counters of the global prediction tables that have been included in the summation are updated only when the overall prediction is wrong or the absolute value of the sum is less than a certain threshold. Their results show

that combining multiple partial matches provides higher prediction accuracy than a single partial match, decreasing the average misprediction rate to 3.41%. A first important difference between the approach presented in [Gao06] and our *branch difference prediction by combining multiple partial matches* developed in paragraph 4.5.3 is that we are focalizing on the unbiased branches identified in our previous work [Gel06, Vin06] instead of "not fully biased" branches. The authors defined a "fully biased" branch being a branch in a certain dynamic context having set its attached bias counter to a maximum value (the counter is incremented each time that branch has a biased behavior and decremented otherwise). Probably it would be better to say "highly biased" branch instead of "fully biased", meaning that it was highly biased (maximum counter) during the "last" processing period (maximum counter at the current prediction moment). However, the main difference is that they used global branch history, while we used local branch difference history. Another important difference consists in how the multiple Markov predictions are combined: we used majority vote (more efficient for our approach) instead of the adder tree used by Gao and Zhou.

In [Sri06] the authors proposed a hybrid branch prediction scheme that employs two PPM predictors, one predicts based on local branch history and the other predicts based on global branch history. For both the local and global PPM predictors, if the local and, respectively, global history were not matched, then shorter patterns are searched, and so on, until a match is found. When a pattern match occurs, the outcome of the branch that succeeded the pattern during its last occurrence is returned as prediction. The two independent predictions are combined through a perceptron. The output of the perceptron is computed as $Y=W_0 + W_1P_L + W_2P_G$, where the inputs P_L and P_G corresponds to the predictions generated by the local and, respectively, global PPM predictor (-1 if not taken and +1 if taken). The final prediction is *taken* if the output Y is positive and *not taken* if Y is negative. The table of weights is indexed by the lower 20 bits of the branch's PC. The perceptron is updated by incrementing the weights whose inputs match the branch outcome and decrementing those with mismatch. The *Neuro-PPM* branch predictor achieves an average misprediction rate of 3%.

3. Finding Difficult-to-Predict Branches

Our first goal is to find the difficult predictable branches in the SPEC2000 benchmarks [SPEC]. As we already pointed out, we consider that a branch in a certain context is difficult predictable if it is unbiased – meaning that the branch behavior (Taken/Not Taken) is not sufficiently polarized for that certain context (local branch history, global history, etc.) – and the taken and not taken outcomes are shuffled. The second goal is to improve prediction accuracy for branches with low polarization rate, introducing new feature sets that will increase their polarization rate and, therefore, their predictability.

3.1. Methodology of Identifying Unbiased Branches

A feature is the binary context on p bits of prediction information such as local history, global history or path. Each static branch finally has associated k dynamic contexts in which it can appear ($k \leq 2^p$). A context instance is a dynamic branch executed in the respective context. We introduce the polarization index (P) of a certain branch context as follows:

$$P(S_i) = \max(f_0,\ f_1) = \begin{cases} f_0, & f_0 \geq 0.5 \\ f_1, & f_0 < 0.5 \end{cases} \qquad (3.1)$$

where:

- $S = \{S_1,\ S_2,\ ...,\ S_k\}$ = set of distinct contexts that appear during all branch instances;
- k = number of distinct contexts, $k \leq 2^p$, where p is the length of the binary context;
- $f_0 = \dfrac{T}{T + NT}$, $f_1 = \dfrac{NT}{T + NT}$, NT = number of "not taken" branch instances corresponding to context S_i, T = number of "taken" branch

instances corresponding to context S_i, $(\forall) i = 1, 2, ..., k$, and obviously $f_0 + f_1 = 1$;

- if $P(S_i) = 1, (\forall) i = 1, 2, ..., k$, then the context S_i is completely biased (100%), and thus, the afferent branch is highly predictable;
- if $P(S_i) = 0.5, (\forall) i = 1, 2, ..., k$, then the context S_i is totally unbiased, and thus, the afferent branch is not predictable if the taken and not taken outcomes are shuffled.

If the taken and respectively not taken outcomes are grouped separately, even in the case of a low polarization index, the branch is predictable. The unbiased branches are not predictable only if the taken and not taken outcomes are shuffled, because in this case, the predictors cannot learn their behavior. For this study we introduce the distribution index for a certain branch context, defined as follows:

$$D(S_i) = \begin{cases} 0, & n_t = 0 \\ \dfrac{n_t}{2 \cdot \min(NT, T)}, & n_t > 0 \end{cases} \qquad (3.2)$$

where:

- n_t = the number of branch outcome transitions, from taken to not taken and vice-versa, in context S_i;
- $2 \cdot \min(NT, T)$ = maximum number of possible transitions;
- k = number of distinct contexts, $k \leq 2^p$, where p is the length of the binary context;
- if $D(S_i) = 1, (\forall) i = 1, 2, ..., k$, then the behavior of the branch in context S_i is "contradictory" (the most unfavorable case), and thus its learning is impossible;
- if $D(S_i) = 0, (\forall) i = 1, 2, ..., k$, then the behavior of the branch in context S_i is constant (the most favorable case), and it can be learned.

As it can be observed in Figure 3.1, we want to systematically analyze different feature sets used by different present-day branch predictors

in order to find and, hopefully, to reduce the list of unbiased branch contexts (contexts with low polarization P).

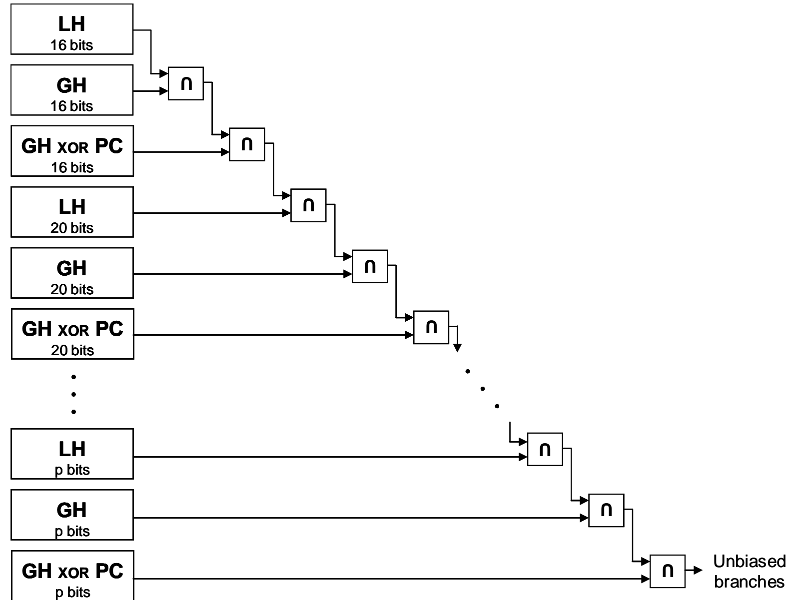

Figure 3.1. Reducing the number of unbiased branches through feature set extension.

We approached an iterative methodology: a certain Feature Set is evaluated only on the unbiased branches determined with the previous Feature Sets, because the rest were solved with the previously considered Feature Sets. Gradually this list is shortened by increasing the lengths of Feature Sets and reapplying the algorithm. Thus, the final list of unbiased branches contains only the branches that were unbiased for all their contexts. The contexts' lengths were varied from 16 bits to 28 bits. For the final list of unbiased branches we will try to find new relevant feature sets in order to further improve their polarization index and, therefore, the prediction accuracy.

This approach is more efficient than one which repeats each time the algorithm on all branches. Beside producing some unpleasant aspects related to simulation time (days / benchmark) and memory (gigabytes of memory needed), the second method would prove even not very accurate. This is because some of the branches that are not solved by a long context can be solved by a shorter one. Through our iterative approach we avoided the occurrence of false problems extending the context.

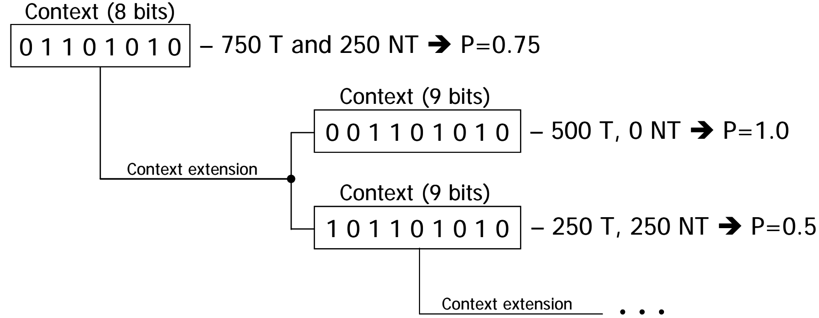

Figure 3.2. The goal of context extension.

Figure 3.2 presents a suggestive example on how unbiased branch contexts can be solved through their extension. We considered that a branch context is unbiased if its polarization index (see relation (3.1)) is less than 0.95. The branch contexts with polarization greater than 0.95 are predictable and will obtain relatively high prediction accuracies (around 95%). More details are presented in paragraph 3.2.4 on a real example from the Stanford *Perm* benchmark [Flo07].

In our experiments we concentrated only on benchmarks with a percentage of unbiased branch context instances (obtained with relation (3.3)), greater than a certain threshold (T=1%) considering that the potential prediction accuracy improvement is not significant in the case of benchmarks with percentage of unbiased context instances less than 1%. If the percentage of unbiased branch contexts is 1%, if they would be solved, the prediction accuracy would increase with maximum 1%. This maximum can be reached when all discovered difficult predictable branches in this stage are solved by the predictor.

$$T = \frac{NUB_i}{NB_i} = 0.01 \qquad (3.3)$$

where NUB_i is the total number of unbiased branch context instances on benchmark *i*, and NB_i is the number of dynamic branches on benchmark *i* (therefore, the total number of branch context instances).

3.2. Experimental Results

All simulation results are reported on 1 billion dynamic instructions, skipping the first 300 million instructions. We note with LH(p)-GH(q)-

GHPC(r) branches unbiased on local history (LH) of p bits, global history (GH) of q bits, and global history XOR-ed by branch address (GHPC) on r bits. In the same manner, for all feature set extensions simulated in this work, LH(p)-GH(q)-GHPC(r)→F(s) denotes that we measure the polarization rate using feature F on s bits (if the feature is the local history, global history or global history XOR-ed by branch address) and/or on s PCs (in the case of path), evaluating only the branches unbiased for local history of p bits, global history of q bits, and global history XOR-ed by branch address on r bits.

3.2.1. Pattern-based Correlation

We started our study evaluating the branch contexts from SPEC2000 benchmarks [SPEC] on local branch history of 16 bits: LH(0)-GH(0)-GHPC(0)→LH(16). In Table 3.1, for each benchmark we presented the percentages of branch contexts with polarization indexes belonging to five different intervals. The column *Dynamic Branches* contains the number of all dynamic conditional branches for each benchmark. The column *Static Br.* contains the number of static branches for each benchmark. For each benchmark we generated using relation (3.1) a list of unbiased branch contexts, having polarization less than 0.95. We considered that the branch contexts with polarization greater than 0.95 are predictable and will obtain relatively high prediction accuracies (around 0.95), therefore, in these cases we considered that the potential improvement of the prediction accuracy is quite low.

SPEC 2000	Dynamic Branches	Static Br.	Polarization Rate (P) [%]					Unbiased Context Instances (P<0.95)	
			[0.5, 0.6)	[0.6, 0.7)	[0.7, 0.8)	[0.8, 0.9)	[0.9, 1.0]		
mcf	118321124	370	10.06	10.50	8.17	8.52	62.74	6812313	5.76%
parser	85382841	1777	6.67	5.90	3.68	4.56	79.19	17589658	20.60%
bzip	42591123	211	15.86	16.50	8.58	6.94	52.12	11252986	26.42%
gzip	71504537	136	15.08	15.63	11.03	9.50	48.76	27692102	38.73%
twolf	70616018	239	14.49	12.72	6.92	5.34	60.54	31763071	44.98%
gcc	90868660	17248	3.06	2.68	1.72	2.30	90.24	9809360	10.80%
Mean	79880717	3330	10.87	10.65	6.68	6.19	65.59	17486582	24.55%

Table 3.1. Polarization rates of branch contexts on local history of 16 bits.

The column *Unbiased Context Instances* contains – for each benchmark – the number of unbiased context instances and respectively the percentage of unbiased context instances reported to all context instances (dynamic

branches). As it can be observed in Table 3.1, the relatively high percentages of unbiased branches (at average 24.55%) show high improvement potential from the predictability point of view.

We continue our work analyzing a global branch history of 16 bits only on the local branch contexts that we already found unbiased for local branch history (see Table 3.1 – last column). In other words, we used a dynamic branch in our evaluations only if its 16 bit local context is one of the unbiased local contexts: LH(16)-GH(0)-GHPC(0)→GH(16). In Table 3.2, for each benchmark we presented the percentages of branch contexts with polarization indexes belonging to five different intervals. The column *Simulated Dynamic Branches* contains the number of evaluated dynamic branches (LH(16)-GH(0)-GHPC(0)) and respectively their percentages reported to all dynamic branches. The column *Simulated St. Br.* represents the number of static branches evaluated within each benchmark. For each benchmark we generated using relation (3.1) a list of unbiased branch contexts on local and global history of 16 bits (LH(16)-GH(16)-GHPC(0)), having polarization less than 0.95. The last column contains the number of unbiased branch context instances and respectively their percentages reported to all dynamic branches. Analyzing comparatively Tables 3.1 and 3.2, we observe that the global branch history reduced the average percentage of unbiased branch context instances from 24.55% to 17.48%.

SPEC 2000	Simulated Dynamic Branches		Simu-lated St. Br.	Polarization Rate (P) [%]					Unbiased Context Instances (P<0.95)	
				[0.5, 0.6)	[0.6, 0.7)	[0.7, 0.8)	[0.8, 0.9)	[0.9, 1.0]		
mcf	6812313	5.76%	25	14.57	11.94	9.25	8.13	56.10	3887052	3.28%
parser	17589658	20.60%	707	6.87	6.98	5.71	6.18	74.26	11064817	12.95%
bzip	11252986	26.42%	83	19.34	16.62	14.36	13.80	35.88	9969701	23.40%
gzip	27692102	38.73%	62	8.98	10.09	9.01	10.88	61.04	20659305	28.89%
twolf	31763071	44.98%	132	8.46	7.43	6.39	9.89	67.83	22893014	32.41%
gcc	9809360	10.80%	4923	4.02	4.13	3.14	3.56	85.15	3563776	3.92%
Mean	17486582	24.55%	988	10.37	9.53	7.97	8.74	63.37	12006278	17.48%

Table 3.2. Polarization rates of branch contexts on global history of 16 bits evaluating only the unbiased local branch contexts of 16 bits.

The next feature set we analyzed is the XOR between a global branch history of 16 bits and the lower part of branch address (PC bits 18÷3): LH(16)-GH(16)-GHPC(0)→GHPC(16). We used again only the branch contexts we found unbiased for the previous feature sets (local and global branch history of 16 bits). In other words, we used a dynamic branch in our evaluations only if its 16 bit local context is one of the unbiased local

contexts (Table 3.1), and its 16 bit global context is one of the unbiased global contexts (Table 3.2). In Table 3.3, for each benchmark we presented the percentages of branch contexts with polarization indexes belonging to five different intervals. For each benchmark we generated again using relation (3.1), a list of unbiased branch contexts with polarization less than 0.95 (LH(16)-GH(16)-GHPC(16)).

SPEC 2000	Simulated Dynamic Branches		Simu-lated St. Br.	Polarization Rate (P) [%]					Unbiased Context Instances (P<0.95)	
				[0.5, 0.6)	[0.6, 0.7)	[0.7, 0.8)	[0.8, 0.9)	[0.9, 1.0]		
mcf	3887069	3.28%	19	30.78	25.21	19.54	17.17	7.30	3887050	3.28%
parser	11065068	12.95%	504	23.84	24.27	19.87	21.56	10.46	11063791	12.95%
bzip	9969757	23.40%	76	28.45	24.43	21.12	20.30	5.70	9969678	23.40%
gzip	20659343	28.89%	51	20.34	22.85	20.43	24.66	11.72	20659290	28.89%
twolf	22893103	32.41%	112	21.11	18.53	15.93	24.69	19.75	22892985	32.41%
gcc	3565197	3.92%	2642	24.05	24.93	18.93	21.46	10.63	3561998	3.91%
Mean	12006590	17.48%	567	24.76	23.37	19.30	21.64	10.92	12005798	17.47%

Table 3.3. Polarization rates on the XOR between global history and branch address on 16 bits evaluating only the unbiased local and global branch contexts of 16 bits.

The last column contains for each benchmark the number of unbiased branch context instances and respectively their percentages reported to all dynamic branches. The high percentages of unbiased branch context instances in the case of bzip, gzip and twolf benchmarks represent a potential improvement of prediction accuracy.

For the determined unbiased branch contexts we are analyzing now if the taken and respectively not taken outcomes are grouped separately. This is necessary, because if the branch outcomes are not shuffled they are predictable using corresponding two-level adaptive predictors, but if these outputs are shuffled the branches are not predictable. We used relation (3.2) in order to determine the distribution indexes for each unpredictable branch context per benchmark. We evaluated only the unbiased dynamic branches obtained using all their contexts of 16 bits (LH(16)-GH(16)-GHPC(16)). Table 3.4 shows for each benchmark the percentages of branch contexts with distribution indexes belonging to five different intervals in the case of local branch history. In the same way, Tables 3.5 and 3.6 present the distribution indexes in the case of global history and respectively the XOR between global history and branch address.

Tables 3.4, 3.5 and 3.6 show that in the case of unbiased branch contexts, the taken and respectively not taken outcomes are not grouped separately, more, they are highly shuffled.

SPEC 2000	Simulated Dynamic Branches		Simu-lated St. Br.	Distribution Rate (D) [%]				
				[0, 0.2)	[0.2, 0.4)	[0.4, 0.6)	[0.6, 0.8)	[0.8, 1.0]
mcf	3887069	3.28%	19	9.21	11.02	46.30	13.32	20.15
parser	11064250	12.95%	483	20.23	9.50	42.44	9.63	18.19
bzip	9969752	23.40%	75	6.78	6.45	44.00	16.80	25.98
gzip	20659339	28.89%	51	5.10	5.38	38.70	20.98	29.85
twolf	22893094	32.41%	110	14.63	5.81	43.42	16.71	19.43
gcc	3564489	3.91%	2553	39.07	9.11	33.32	6.00	12.50
Mean	12006332	17.47%	548	15.83	7.87	41.36	13.90	21.01

Table 3.4. Distribution rates on local history of 16 bits evaluating only the branches that were unbiased on all their 16 bit contexts (on local history, global history and respectively XOR of global history and branch address).

SPEC 2000	Simulated Dynamic Branches		Simu-lated St. Br.	Distribution Rate (D) [%]				
				[0, 0.2)	[0.2, 0.4)	[0.4, 0.6)	[0.6, 0.8)	[0.8, 1.0]
mcf	3887069	3.28%	19	0.27	4.30	37.75	34.38	23.31
parser	11064250	12.95%	483	6.92	14.62	36.63	19.33	22.50
bzip	9969752	23.40%	75	0.25	2.94	32.24	37.43	27.13
gzip	20659339	28.89%	51	0.26	2.18	26.45	35.19	35.91
twolf	22893094	32.41%	110	0.84	5.12	26.84	28.44	38.75
gcc	3564489	3.91%	2553	8.10	18.03	38.66	16.06	19.15
Mean	12006332	17.47%	548	2.77	7.86	33.09	28.47	27.79

Table 3.5. Distribution rates on global history of 16 bits evaluating only the branches that have all their 16 bit contexts unbiased (on local history, global history and respectively XOR of global history and branch address).

SPEC 2000	Simulated Dynamic Branches		Simu-lated St. Br.	Distribution Rate (D) [%]				
				[0, 0.2)	[0.2, 0.4)	[0.4, 0.6)	[0.6, 0.8)	[0.8, 1.0]
mcf	3887069	3.28%	19	0.27	4.30	37.75	34.38	23.31
parser	11064250	12.95%	483	6.92	14.62	36.63	19.33	22.50
bzip	9969752	23.40%	75	0.25	2.94	32.24	37.43	27.13
gzip	20659339	28.89%	51	0.26	2.18	26.45	35.19	35.91
twolf	22893094	32.41%	110	0.84	5.12	26.84	28.44	38.75
gcc	3564489	3.91%	2553	8.10	18.03	38.66	16.06	19.15
Mean	12006332	17.47%	548	2.77	7.86	33.09	28.47	27.79

Table 3.6. Distribution rates on the XOR between global history and branch address on 16 bits evaluating only branches having all 16 bit contexts unbiased (on local and global history and the XOR of global history and branch address).

The percentage of unbiased branch contexts having highly shuffled outcomes (with distribution index greater than 0.4) is 76.3% in the case of local history of 16 bits (see Table 3.4), 89.37% in the case of global history of 16 bits (see Table 3.5), and 89.37% in the case of global history XOR-ed by branch address on 16 bits (see Table 3.6). We obtained the same distribution indexes for both the global history and respectively the XOR between global history and branch address (Tables 3.5 and 3.6).

A distribution index of 1.0 means the highest possible alternation frequency (with taken or not taken periods of 1). A distribution index of 0.5 means again a high alternation, since, supposing a constant frequency, the taken or not taken periods are only 2, lower than the predictors' learning times. In the same manner, periods of 3 introduce a distribution of about 0.25, and periods of 5 generate a distribution index of 0.15, therefore we considered that if the distribution index is lower than 0.2 the taken and not taken outcomes are not shuffled, and the branch's behavior can be learned.

We continued our evaluations extending the lengths of feature sets from 16 bits to 20, 24 and respectively 28 bits, our hypothesis being that the longer feature sets will increase the polarization index and, therefore, the prediction accuracy. We started with a local branch history of 20 bits (Table 3.7), evaluating again only the branch contexts we found unbiased for the previous feature sets of 16 bits: LH(16)-GH(16)-GHPC(16)→LH(20).

SPEC 2000	**Simulated Dynamic Branches**		**Simu-lated St. Br.**	**Polarization Rate (P) [%]**					**Unbiased Context Instances (P<0.95)**	
				[0.5, 0.6)	**[0.6, 0.7)**	**[0.7, 0.8)**	**[0.8, 0.9)**	**[0.9, 1.0]**		
mcf	3887050	3.28%	19	8.41	7.96	5.28	5.97	72.37	3147989	2.66%
parser	11063878	12.95%	476	8.50	6.70	3.87	4.44	76.49	7838166	9.18%
bzip	9969651	23.40%	75	8.93	4.69	2.10	2.17	82.11	6493881	15.24%
gzip	20659242	28.89%	51	9.98	7.47	4.55	4.84	73.16	17753722	24.82%
twolf	22892904	32.41%	110	12.79	10.91	5.17	3.93	67.20	17540719	24.83%
gcc	3563213	3.91%	2546	7.79	6.31	3.68	4.56	77.66	2061136	2.26%
Mean	12005990	17.47%	546	9.40	7.34	4.10	4.31	74.83	9139269	13.17%

Table 3.7. Polarization rates on local history of 20 bits evaluating only the branches that have all their 16 bit contexts unbiased (on local history, global history and respectively XOR of global history and branch address).

The column *Polarization Rate* from Table 3.7 presents the percentages of branch contexts with polarization indexes belonging to five different intervals. The last column of Table 3.7 shows for each benchmark the number of unbiased dynamic branches (LH(20)-GH(16)-GHPC(16)), and respectively their percentage reported to all dynamic branches.

Table 3.8 shows the results obtained using a global branch history of 20 bits: LH(20)-GH(16)-GHPC(16)→GH(20). The last column of Table 3.8 shows the number of unbiased dynamic branches (LH(20)-GH(20)-GHPC(16)) and their percentage reported to all dynamic branches.

SPEC 2000	Simulated Dynamic Branches		Simu-lated St. Br.	Polarization Rate (P) [%]					Unbiased Context Instances (P<0.95)	
				[0.5, 0.6)	[0.6, 0.7)	[0.7, 0.8)	[0.8, 0.9)	[0.9, 1.0]		
mcf	3148005	2.66%	18	20.06	20.55	13.08	10.60	35.71	3057312	2.58%
parser	7838384	9.18%	446	15.44	14.61	10.83	11.04	48.09	7166404	8.39%
bzip	6493918	15.24%	74	15.86	17.02	12.45	12.43	42.24	6228047	14.62%
gzip	17753750	24.82%	45	15.32	16.89	15.88	17.75	34.16	17215762	24.07%
twolf	17540776	24.83%	103	13.96	12.79	11.63	17.61	44.00	16240443	22.99%
gcc	2062167	2.26%	2299	14.59	13.77	9.35	9.93	52.37	1767385	1.94%
Mean	9139500	13.17%	497	15.87	15.93	12.20	13.22	42.76	8612559	12.43%

Table 3.8. Polarization rates on global history of 20 bits evaluating only the unbiased branches on local history of 20 bits, global history of 16 bits, and the XOR of global history and branch address on 16 bits.

In the same manner, Table 3.9 shows the results obtained using a XOR of 20 bits between global history and branch address: LH(20)-GH(20)-GHPC(16)→GHPC(20). The last column of Table 3.9 shows for each benchmark the number and percentage of unbiased dynamic branches: LH(20)-GH(20)-GHPC(20).

SPEC 2000	Simulated Dynamic Branches		Simu-lated St. Br.	Polarization Rate (P) [%]					Unbiased Context Instances (P<0.95)	
				[0.5, 0.6)	[0.6, 0.7)	[0.7, 0.8)	[0.8, 0.9)	[0.9, 1.0]		
mcf	3057327	2.58%	18	30.53	31.28	19.91	16.14	2.13	3057309	2.58%
parser	7166723	8.39%	429	27.62	26.16	19.37	19.76	7.08	7166215	8.39%
bzip	6228107	14.62%	73	26.21	28.12	20.57	20.53	4.57	6228010	14.62%
gzip	17215799	24.07%	45	20.78	22.96	21.58	24.13	10.55	17215749	24.07%
twolf	16240535	22.99%	101	21.26	19.48	17.70	26.81	14.74	16240434	22.99%
gcc	1769008	1.94%	2019	28.28	26.84	18.17	19.29	7.41	1766800	1.94%
Mean	8612917	12.43%	447	25.78	25.80	19.55	21.11	7.74	8612420	12.43%

Table 3.9. Polarization rates on the XOR of 20 bits between global history and branch address evaluating only the branches unbiased for local and global history of 20 bits respectively the XOR of global history and branch address on 16 bits.

As it can be observed a considerable number of unbiased branches become biased if the feature sets are extended from 16 bits to 20 bits. Extending the feature set length from 16 bits to 20 bits, the percentage of unbiased dynamic branches decreased at average from 17.47% (see Table 3.3) to

12.43% (Table 3.9). Using the same simulation methodology, we extend the feature sets to 24 bits.

SPEC 2000	Simulated Dynamic Branches		Simu-lated St. Br.	Polarization Rate (P) [%]					Unbiased Context Instances (P<0.95)	
				[0.5, 0.6)	[0.6, 0.7)	[0.7, 0.8)	[0.8, 0.9)	[0.9, 1.0]		
mcf	3057318	2.58%	18	9.04	7.95	4.59	5.41	73.01	2632531	2.22%
parser	7166415	8.39%	424	10.88	8.16	4.19	4.44	72.34	5083585	5.95%
bzip	6228031	14.62%	73	8.41	4.71	2.46	2.84	81.59	4250654	9.98%
gzip	17215734	24.07%	45	9.20	6.19	3.64	4.19	76.78	13753938	19.23%
twolf	16240411	22.99%	101	10.14	5.40	2.21	1.95	80.31	12308193	17.42%
gcc	1768113	1.94%	1980	11.73	9.02	5.11	6.14	68.00	1227407	1.35%
Mean	8612670	12.43%	440	9.90	6.90	3.70	4.16	75.33	6542718	9.36%

Table 3.10. Polarization rates on local history of 24 bits only for branches that were unbiased on all their 20 bit contexts (on local history, global history and respectively XOR of global history and branch address).

Table 3.10 shows the results obtained using a local branch history of 24 bits: LH(20)-GH(20)-GHPC(20)→LH(24). The last column of Table 3.10 shows for each benchmark the number and percentage of unbiased dynamic branches: LH(24)-GH(20)-GHPC(20).

Table 3.11 shows the results obtained using a global branch history of 24 bits: LH(24)-GH(20)-GHPC(20)→GH(24). The last column of Table 3.11 shows the number of unbiased dynamic branches (LH(24)-GH(24)-GHPC(20)) and their percentage reported to all dynamic branches.

SPEC 2000	Simulated Dynamic Branches		Simu-lated St. Br.	Polarization Rate (P) [%]					Unbiased Context Instances (P<0.95)	
				[0.5, 0.6)	[0.6, 0.7)	[0.7, 0.8)	[0.8, 0.9)	[0.9, 1.0]		
mcf	2632542	2.22%	18	15.20	13.79	7.13	5.90	57.98	2568911	2.17%
parser	5083795	5.95%	414	18.82	16.61	10.90	10.41	43.25	4664394	5.46%
bzip	4250689	9.98%	73	12.10	11.31	7.12	7.60	61.87	3799893	8.92%
gzip	13753960	19.23%	44	18.43	18.17	15.37	16.36	31.67	13480788	18.85%
twolf	5459637	17.42%	93	16.99	14.90	10.91	13.88	43.32	5144339	7.28%
gcc	1228364	1.35%	1856	17.16	14.61	9.94	10.15	48.14	1097445	1.20%
Mean	5401498	9.36%	416	16.45	14.89	10.22	10.71	47.70	5125962	7.31%

Table 3.11. Polarization rates on global history of 24 bits evaluating only the branches unbiased for local history of 24 bits, global history of 20 bits and respectively XOR of global history and branch address on 20 bits.

Table 3.12 presents the results obtained using the XOR between global branch history and branch address on 24 bits: LH(24)-GH(24)-

GHPC(20)→GHPC(24). The last column of Table 3.12 shows for each benchmark the number and percentage of unbiased dynamic branches: LH(24)-GH(24)-GHPC(24). Extending the feature set length from 20 bits to 24 bits, the percentage of unbiased dynamic branches decreased at average from 12.43% (see Table 3.9) to 7.31% (Table 3.12).

SPEC 2000	**Simulated Dynamic Branches**		**Simu-lated St. Br.**	**Polarization Rate (P) [%]**					**Unbiased Context Instances (P<0.95)**	
				[0.5, 0.6)	**[0.6, 0.7)**	**[0.7, 0.8)**	**[0.8, 0.9)**	**[0.9, 1.0]**		
mcf	2568928	2.17%	18	35.55	32.24	16.67	13.79	1.75	2568910	2.17%
parser	4664693	5.46%	398	31.21	27.52	18.08	17.25	5.93	4664273	5.46%
bzip	3799936	8.92%	72	30.43	28.45	17.91	19.13	4.07	3799859	8.92%
gzip	13480825	18.85%	41	24.64	24.29	20.55	21.87	8.66	13480783	18.85%
twolf	5144419	7.28%	89	27.03	23.73	17.38	22.10	9.76	5144327	7.28%
gcc	1098795	1.20%	1668	30.73	26.27	17.87	18.39	6.75	1097009	1.20%
Mean	5126266	7.31%	381	29.93	27.08	18.07	18.75	6.15	5125860	7.31%

Table 3.12. Polarization rates on the XOR of 24 bits between global history and branch address evaluating only the branches unbiased for local history of 24 bits, global history of 24 bits and XOR of global history and branch address on 20 bits.

We extended again the feature sets to 28 bits. Table 3.13 shows the results obtained using a local branch history of 28 bits: LH(24)-GH(24)-GHPC(24)→LH(28). The last column of Table 3.13 shows for each benchmark the number of unbiased dynamic branches (LH(28)-GH(24)-GHPC(24)) and their percentage reported to all dynamic branches.

SPEC 2000	**Simulated Dynamic Branches**		**Simu-lated St. Br.**	**Polarization Rate (P) [%]**					**Unbiased Context Instances (P<0.95)**	
				[0.5, 0.6)	**[0.6, 0.7)**	**[0.7, 0.8)**	**[0.8, 0.9)**	**[0.9, 1.0]**		
mcf	2568923	2.17%	18	10.62	8.64	4.69	5.35	70.69	2174101	1.83%
parser	4664502	5.46%	395	11.17	7.09	3.72	4.07	73.95	3301587	3.86%
bzip	3799904	8.92%	71	10.16	5.90	3.04	3.59	77.30	2728593	6.40%
gzip	13480777	18.85%	41	9.76	6.14	3.50	4.14	76.46	10691142	14.95%
twolf	5144325	7.28%	87	9.03	4.44	2.81	3.76	79.96	4208376	5.95%
gcc	~~1098269~~	~~1.20%~~	~~1644~~	~~13.68~~	~~10.29~~	~~5.68~~	~~6.76~~	~~63.59~~	~~774654~~	~~0.85%~~
Mean	5931686	8.54%	122	10.14	6.44	3.55	4.18	75.67	4620759	6.60%

Table 3.13. Polarization rates on local history of 28 bits only for branches that were unbiased on all their 24 bit contexts (on local history, global history and respectively XOR of global history and branch address).

As it can be observed, in the case of the gcc benchmark, extending the feature set length to 28 bits, the percentage of the unbiased context instances

is less than the threshold T of 1% (see relation (3.3)), and thus we eliminate it from our next evaluations. We consider that the conditional branches from the gcc benchmark are not difficult predictable using feature lengths of 28 bits. As a consequence the results obtained with the *gcc* benchmark are not included in the average results from Table 3.13.

SPEC 2000	Simulated Dynamic Branches		Simulated St. Br.	Polarization Rate (P) [%]					Unbiased Context Instances (P<0.95)	
				[0.5, 0.6)	[0.6, 0.7)	[0.7, 0.8)	[0.8, 0.9)	[0.9, 1.0]		
mcf	2174117	1.83%	18	15.41	11.53	6.18	5.29	61.60	2149108	1.81%
parser	3301768	3.86%	370	21.26	17.06	10.39	10.18	41.11	3041426	3.56%
bzip	2728627	6.40%	69	11.81	8.86	5.07	5.55	68.72	2280197	5.35%
gzip	10691161	14.95%	41	19.36	17.05	13.50	14.84	35.25	10405692	14.55%
twolf	4208418	5.95%	85	16.53	14.43	10.21	13.55	45.29	4007088	5.67%
Mean	4620818	6.60%	116	16.87	13.78	9.07	9.88	50.39	4376702	6.19%

Table 3.14. Polarization rates on global history of 28 bits evaluating only the branches unbiased for local history of 28 bits, global history of 24 bits and respectively the XOR of global history and branch address on 24 bits.

Table 3.14 presents the results obtained when we used a global branch history of 28 bits: LH(28)-GH(24)-GHPC(24)→GH(28). The column *Unbiased Context Instances* from Table 3.14 presents for each benchmark the number and percentage of unbiased dynamic branches: LH(28)-GH(28)-GHPC(24).

Finally, Table 3.15 shows the results obtained using the XOR of global branch history and branch address on 28 bits: LH(28)-GH(28)-GHPC(24)→GHPC(28). The last column of Table 3.15 shows for each benchmark the number of unbiased dynamic branches (LH(28)-GH(28)-GHPC(28)) and their percentage reported to all dynamic branches.

SPEC 2000	Simulated Dynamic Branches		Simulated St. Br.	Polarization Rate (P) [%]					Unbiased Context Instances (P<0.95)	
				[0.5, 0.6)	[0.6, 0.7)	[0.7, 0.8)	[0.8, 0.9)	[0.9, 1.0]		
mcf	2149125	1.81%	18	39.26	29.37	15.73	13.46	2.17	2149107	1.81%
parser	3041691	3.56%	357	34.21	27.48	16.71	16.39	5.22	3041301	3.56%
bzip	2280240	5.35%	69	36.29	27.22	15.57	17.05	3.87	2280161	5.35%
gzip	10405726	14.55%	41	27.56	24.28	19.22	21.13	7.81	10405684	14.55%
twolf	4007152	5.67%	82	27.73	24.21	17.12	22.73	8.21	4007068	5.67%
Mean	4376787	6.19%	113	33.01	26.51	16.87	18.15	5.45	4376664	6.19%

Table 3.15. Polarization rates on the XOR of 28 bits between global history and branch address evaluating only the branches unbiased for local and global history of 28 bits respectively the XOR of global history and branch address on 24 bits.

Extending the feature set length from 24 bits to 28 bits, the percentage of unbiased dynamic branches decreased at average from 7.31% (see Table 3.12) to 6.19% (see Table 3.15). Despite of the feature set extension, the number of unbiased dynamic branches remains still high (6.19%), and thus, it is obvious that using longer feature sets is not sufficient.

The global history solves at average 2.56% of the unbiased dynamic branches not solved with local history (see Figure 3.3). The hashing between global history and branch address (XOR) behaves just like the global history, and it does not improve further the polarization rate. In Figure 3.3 can be also observed that increasing the branch history, the percentage of unbiased dynamic branches decreases, suggesting a correlation between branches situated at a large distance in the dynamic instruction stream. The results also show that the "ultimative predictibility limit" of history context-based prediction is approximatively 94%, considering unbiased branches as completely unpredictable. A conclusion based on our simulation methodology is that 94% of dynamic branches can be solved with prediction information of up to 28 bits (some of them are solved with 16 bits, others with 20, 24 or 28 bits).

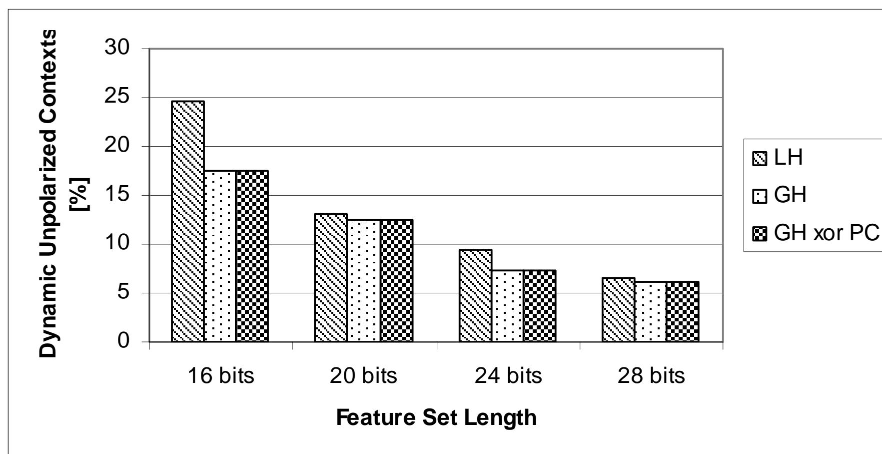

Figure 3.3. Reduction of average percentages of unbiased context instances (P<0.95) by extending the lengths of feature sets.

In another work we have studied the polarization of branches but using a little different simulation methodology [Oan06]. We evaluated local history concatenated with global history. The simulation methodology is presented in Figure 3.4.

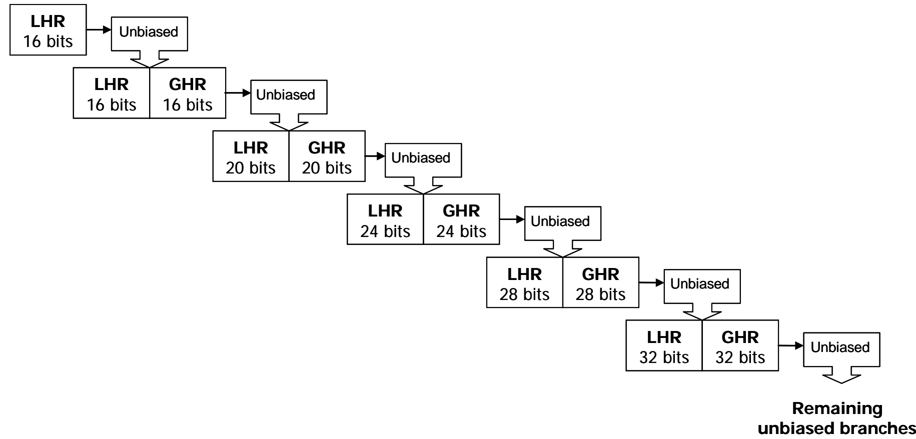

Figure 3.4. Identifying unbiased branches by using the local history concatenated with the global history.

The evaluation results presented in Table 3.16 show that these longer contexts, due to their better precision, have higher polarization index. Comparing our results, it is obvious that a certain feature set LH(p)-GH(p) from Table 3.16 is approximatively equivalent in terms of polarization rate with feature set GH(p+4) from Tables 3.8, 3.11 and 3.14. In other words, the same percentage of unbiased context instances is obtained for both LH(p)-GH(p) and GH(p+4) feature sets, but the number of bits in the correlation information is different: (p+p) bits of local and global history, and respectively (p+4) bits of global history.

Benchmark	**LH(0)-GH(0) ->LH(16)-GH(0)**	**LH(16)-GH(0) ->LH(16)-GH(16)**	**LH(16)-GH(16) ->LH(20)-GH(20)**	**LH(20)-GH(20) ->LH(24)-GH(24)**	**LH(24)-GH(24) ->LH(28)-GH(28)**	**LH(28)-GH(28) ->LH(32)-GH(32)**
bzip	26.42%	12.83%	7.53%	4.70%	3.08%	2.10%
gzip	38.73%	24.58%	17.84%	12.67%	9.12%	6.16%
mcf	5.76%	3.09%	2.44%	2.09%	1.78%	1.49%
parser	20.61%	7.42%	4.77%	3.01%	1.98%	1.40%
twolf	44.98%	23.94%	12.79%	8.28%	5.70%	3.90%
gcc	10.85%	2.50%	1.41%	~~0.88%~~	~~0.58%~~	~~0.39%~~
Average	24.56%	12.39%	7.80%	6.15%	4.33%	3.01%

Table 3.16. The percentages of unbiased context instances, after each context length extension, obtained by using only the local history concatenated with the global history.

Taking into account that increasing the prediction accuracy with 1%, the IPC (instructions-per-cycle) is improved with more than 1% (it grows non-linearly) [Yeh92], there are great chances to obtain considerably better overall performances even if not all of the 6.19% difficult predictable branches will be solved. Therefore, we consider that this 6.19% represents a significant percentage of unbiased branch context instances, and in the same time a good improvement potential in terms of prediction accuracy and IPC. Focalising on these unbiased branches – in order to design some efficient path-based predictors for them [Nair95, Vin99b] – the overall prediction accuracy should increase with some percents, that would be quite remarkable. The simulation results also lead to the conclusion that as higher is the feature set length used in the prediction process, as higher is the branch polarization index and hopefully the prediction accuracy (Figure 3.3). A certain large context (e.g. 100 bits) – due to its better precision – has lower occurrence probability than a smaller one, and higher dispersion capabilities (the dispersion grows exponentially). Thus, very large contexts can significantly improve the branch polarization and the prediction accuracy too. However, they are not always feasable for hardware implementation. The question is: what feature set length is really feasable for hardware implementation, and more important, in this case, which is the solution regarding the unbiased branches? In our opinion, as we'll further show, a feasable solution in this case could be given by path-predictors.

3.2.2. Path-based Correlation

The path information could be a solution for relatively short history contexts (low correlations). Our hypothesis is that short contexts used together with path information should replace significantly longer contexts, providing the same prediction accuracy. A common criticism for most of the present two-level adaptive branch prediction schemes consists in the fact that they used insufficient global correlation information [Vin99b]. There are situations when a certain static branch, in the same global history context pattern, has different behaviors (taken/not taken), and therefore the branch in that context is unbiased. If each bit belonging to the global history will be associated during the prediction process with its corresponding PC, the context of the current branch becomes more precisely, and therefore its prediction accuracy could be better. Our next goal is to extend the correlation information with the path, according to the above idea [Vin99b]. Extending the correlation information in this way, suggests that at different

occurrences of a certain static branch with the same global history context, the path contexts can be different.

We started our evaluations regarding the path, studying the gain obtained by introducing paths of different lengths. The analyzed feature consists of a global branch history of 16 bits and the last *p* PCs. We applied this feature only to dynamic branches that we already found unbiased (P<0.95) for local and global history of 16 bits and respectively global history XOR-ed by branch address on 16 bits.

Benchmark	LH(16)-GH(16)-GHPC(16)	LH(16)-GH(16)-GHPC(16) ->PATH(1)	LH(16)-GH(16)-GHPC(16) ->PATH(16)	LH(16)-GH(16)-GHPC(16) ->PATH(20)	LH(16)-GH(16)-GHPC(16) ->LH(20)
bzip	23.40%	23.35%	22.16%	20.38%	15.24%
gzip	28.89%	28.88%	28.17%	27.51%	24.82%
mcf	3.28%	3.28%	3.28%	3.20%	2.66%
parser	12.95%	12.89%	12.01%	10.95%	9.18%
twolf	32.41%	32.41%	31.46%	27.10%	24.83%
gcc	3.91%	3.91%	3.56%	3.02%	2.26%
Average	17.47%	17.45%	16.77%	15.36%	13.17%
Gain		0.02%	0.70%	2.11%	4.30%

Table 3.17. The gain introduced by the path of different lengths (1, 16, 20 PCs) versus the gain introduced by extended local history (20 bits).

Column LH(16)-GH(16)-GHPC(16) from Table 3.17, presents the percentage of unbiased contexts for each benchmark. Columns LH(16)-GH(16)-GHPC(16)→PATH(1), LH(16)-GH(16)-GHPC(16)→PATH(16) and LH(16)-GH(16)-GHPC(16)→PATH(20) presents the percentages of unbiased context instances obtained using a global history of 16 bits and a path of 1, 16 and respectively 20 PCs. The last column presents the percentages of unbiased context instances extending the local history to 20 bits (without path). For each feature is presented the gain opposite to the first column average. It can be observed that a path of 1 introduces a not significant gain of 0.2%. Even a path of 20 introduces a gain of only 2.11% related to the more significant gain of 4.30% introduced by an extended local branch history of 20 bits. The results show (Table 3.17) that the path is useful only in the case of short contexts. Thus, a branch history of 16 bits compresses and approximates well the path information. In other words, a branch history of 16 bits spreads well the different paths that lead to a certain dynamic branch.

We continue our work evaluating – on all branches (non-iterative simulation) – the number of unbiased context instances (P<0.95) using as

prediction information paths of different lengths (*p* PCs) together with global histories of the same lengths (*p* bits).

Bench.	**p=1**	**p=4**	**p=8**	**p=12**	**p=16**	**p=20**	**p=24**
bzip	58.54%	39.00%	37.24%	35.08%	32.41%	31.29%	28.01%
gzip	49.85%	45.93%	43.58%	35.67%	34.10%	33.31%	33.02%
mcf	27.85%	21.30%	6.38%	5.89%	6.35%	5.58%	5.20%
parser	57.75%	44.64%	36.37%	30.63%	27.25%	23.00%	20.03%
twolf	67.49%	59.07%	51.28%	43.51%	37.12%	31.47%	28.47%
gcc	34.17%	26.34%	17.65%	12.61%	9.51%	7.85%	6.64%
Average	49.28%	39.38%	32.08%	27.23%	24.46%	22.08%	20.23%

Table 3.18. The percentages of unbiased context instances using as context only the global history of *p* bits.

Bench.	**p=1**	**p=4**	**p=8**	**p=12**	**p=16**	**p=20**	**p=24**
bzip	38.99%	36.93%	34.41%	32.16%	30.15%	27.52%	23.90%
gzip	48.53%	44.81%	42.20%	34.45%	33.21%	32.73%	32.31%
mcf	26.01%	20.98%	6.23%	5.85%	6.48%	5.57%	5.19%
parser	48.42%	39.50%	32.13%	27.48%	24.66%	20.82%	18.65%
twolf	62.65%	55.68%	49.47%	42.60%	35.81%	30.66%	27.88%
gcc	28.51%	20.42%	13.84%	10.53%	8.44%	7.12%	6.14%
Average	42.19%	36.39%	29.71%	25.51%	23.13%	20.74%	19.01%

Table 3.19. The percentages of unbiased context instances using as feature the global history of *p* bits together with the path of *p* PCs.

The results are presented in Table 3.19, and in Figure 3.5 they are compared with the results obtained using only global history (see Table 3.18). In the case of the 'mcf' benchmark we obtained higher percentage of unbiased context instances when we extended the correlation information (Table 3.19) from 12 bits of global history and 12 PCs (p=12) to 16 bits of global history and 16 PCs (p=16). This growth is possible because a certain biased context ($P \geq 0.95$), through extension is splitted into more subcontexts, and some of these longer contexts can be unbiased ($P<0.95$), thus increasing the number of unbiased branches. Again, the results obtained with long global history patterns (contexts) are closer to those obtained with path patterns of the same lengths, meaning that long global history (*p* bits) approximates very well the longer path information (*p* PCs).

As it can be observed in Figure 3.5, an important gain is obtained through path in the case of short contexts ($p<16$). A branch history of more than 12 bits, compresses well the path information, and therefore, in these cases, the gain introduced by the path is not significant.

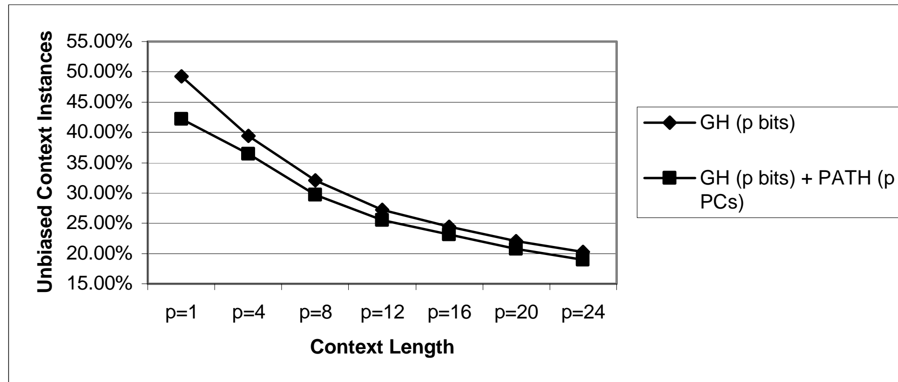

Figure 3.5. The gain introduced by the path for different context lengths – SPEC2000 benchmarks.

Desmet shows in her PhD thesis [Des06] that complete path (all branches) is more efficient than simple path (only conditional branches) from the entropy point of view. This is in contradiction with our results presented in Table 3.20, where we compared these types of path from the unbiased branch percentage point of view. This contradiction can be justified (?) by observing the following differences between our measurements:

- Desmet measured per branch entropy and presented the average entropy, while we measured per branch-context polarization and presented the average percentage of branch contexts having polarization less than 0.95;
- Desmet's path consists in the PCs corresponding to the target instructions (as Nair did), while our path consists in the PCs of branches;
- Desmet uses short histories (p=1, 2, 5 PCs), while our evaluations were generated on a considerable larger interval (p=1, 4, 8, …, 24 PCs).

As we explain below, paradoxically, the simple path is more rich in information than complete path (for the same number of PCs), justifying our results presented in Table 3.20. Let's consider the following sequence of instructions:

... bne1 ... bne2 ... jr ... bne3 ... bne4 ... bne5=?

If we use a path history of 4 PCs (p=4), then:

- simple path = *bne1*, *bne2*, *bne3*, *bne4*;
- complete path = *bne2*, *jr*, *bne3*, *bne4*.

The unconditional branch *jr* brings less information, because it is always *taken*, and therefore, between *bne2* and *bne3* through *jr* only one path is possible, while through conditional branches two paths are possible. Thus, the path consisting exclusively in conditional branches is better than complete path (see Table 3.20).

Context	p=1	p=4	p=8	p=12	p=16	p=20	p=24
GH (p bits)	49.28	39.38	32.08	27.23	24.46	22.08	20.23
GH (p bits) + FullTargetPath (p PCs)	46.74	37.23	30.72	26.50	23.89	21.58	19.88
GH (p bits) + FullPath (p PCs)	43.21	37.03	30.49	26.41	23.86	21.56	19.86
GH (p bits) + CondTargetPath (p PCs)	45.13	36.41	29.76	25.56	23.18	20.77	19.09
GH (p bits) + CondPath (p PCs)	42.19	36.39	29.71	25.51	23.13	20.74	19.01

Table 3.20. Percentages of unbiased branches on the SPEC2000 benchmarks [%].

We also compared the path consisting in PCs of branches with the path consisting in PCs of target instructions. The path of branch PCs is slightly better, however the difference is unsignificant (see Table 3.20).

Further, we present some results obtained applying the same methodology on Branch Prediction World Championship benchmarks – proposed by Intel [CBP, Loh05a]. We continue to evaluate – on all branches using the non-iterative simulation – paths of different lengths (*p* PCs) used together with global histories of the same lengths (*p* bits). The results are presented in Table 3.22, and in Figure 3.6 they are compared with the results obtained using only global history (see Table 3.21).

As it can be observed from Tables 3.21, 3.22 and from Figure 3.6, the results produced (unbiased context instances ratio) by the Intel benchmarks have the same profile like that obtained on the SPEC2000 benchmarks. Actually, rich contexts (long patterns) reduce almost to zero the advantage introduced by using the path information.

Benchmark	p=1	p=4	p=8	p=12	p=16	p=20	p=24	p=28	p=32
dist-fp-1	25.70	21.11	18.45	16.38	14.61	12.64	10.62	9.74	9.46
dist-fp-2	20.47	9.11	8.48	5.18	5.19	5.43	5.36	5.20	5.32
dist-fp-3	3.04	1.82	0.95	0.18	0.00	0.00	0.00	0.00	0.00
dist-fp-4	11.41	8.96	4.58	3.92	3.59	3.00	2.27	1.73	1.37
dist-fp-5	68.91	30.11	16.81	6.09	5.19	5.19	4.46	3.72	3.72
dist-int-1	47.98	40.00	28.97	24.93	20.52	16.39	14.01	10.80	9.38
dist-int-2	55.98	48.24	39.61	32.81	27.46	22.97	19.54	17.19	15.11
dist-int-3	66.26	55.74	47.23	38.35	31.21	26.20	22.74	20.10	17.56
dist-int-4	45.31	42.29	29.53	21.11	16.50	13.77	11.18	9.92	8.89
dist-int-5	2.50	1.48	1.10	0.91	0.78	0.72	0.67	0.67	0.66
dist-mm-1	72.68	66.29	56.09	52.16	47.16	44.32	40.95	37.14	33.04
dist-mm-2	39.43	37.51	33.48	30.65	28.65	26.75	24.79	22.56	20.20
dist-mm-3	19.33	15.62	13.43	11.54	10.20	6.80	6.17	5.42	5.20
dist-mm-4	8.41	6.11	6.86	5.91	5.01	4.24	3.13	3.08	2.98
dist-mm-5	38.16	29.02	22.08	16.97	14.53	12.17	10.64	9.22	7.91
dist-serv-1	16.63	11.60	7.92	6.01	5.21	4.03	3.17	2.75	2.57
dist-serv-2	15.59	11.08	7.66	5.91	4.81	3.76	3.11	2.72	2.44
dist-serv-3	29.87	25.68	20.52	16.84	14.06	12.37	9.02	8.26	7.47
dist-serv-4	15.53	11.04	8.00	7.06	5.86	5.17	4.63	4.22	3.93
dist-serv-5	15.94	11.27	7.95	7.21	6.17	5.38	5.08	4.55	4.15
Average	30.96	24.20	18.99	15.51	13.33	11.56	10.08	8.95	8.07

Table 3.21. The percentages of unbiased context instances using as context only the global history of *p* bits – Intel benchmarks [%].

Benchmark	p=1	p=4	p=8	p=12	p=16	p=20	p=24	p=28	p=32
dist-fp-1	25.72	20.97	17.44	15.56	13.11	11.54	10.03	9.16	8.93
dist-fp-2	20.46	8.92	8.21	5.32	5.33	5.58	5.52	5.41	5.27
dist-fp-3	2.77	1.73	0.86	0.00	0.00	0.00	0.00	0.00	0.00
dist-fp-4	10.86	8.95	4.45	3.78	3.59	2.99	2.26	1.73	1.36
dist-fp-5	65.40	28.47	15.91	5.56	5.19	4.46	3.72	3.72	3.72
dist-int-1	44.02	32.67	26.51	23.05	18.05	15.16	12.40	10.28	8.63
dist-int-2	52.98	42.77	34.33	28.62	24.01	20.79	18.07	16.03	14.25
dist-int-3	64.24	55.42	46.82	38.04	31.10	26.15	22.56	20.00	17.51
dist-int-4	43.98	38.08	26.22	20.29	15.74	12.88	10.87	9.78	8.84
dist-int-5	2.27	1.22	0.93	0.82	0.75	0.71	0.66	0.66	0.65
dist-mm-1	71.98	60.26	50.31	48.23	44.58	41.28	37.59	33.59	29.38
dist-mm-2	36.70	35.11	31.19	29.01	27.52	26.09	24.00	21.65	19.18
dist-mm-3	18.21	14.57	13.09	11.42	9.83	6.76	6.13	5.41	5.20
dist-mm-4	8.33	5.86	6.86	5.90	5.00	4.21	3.12	3.08	2.98
dist-mm-5	35.82	26.83	19.60	15.60	13.74	11.72	10.24	8.85	7.66
dist-serv-1	14.71	9.12	6.57	5.08	4.32	3.37	2.98	2.52	2.19
dist-serv-2	13.85	8.79	6.38	4.74	3.79	3.29	2.75	2.48	2.17
dist-serv-3	27.88	20.43	15.28	14.02	12.50	11.45	8.36	7.61	6.94
dist-serv-4	13.77	9.03	6.88	6.16	5.43	4.82	4.51	4.08	3.74
dist-serv-5	14.16	9.42	6.77	6.47	5.74	5.16	4.93	4.38	3.91
Average	29.41	21.93	17.23	14.38	12.46	10.92	9.53	8.52	7.63

Table 3.22. The ratio of unbiased context instances using as features the global history of *p* bits together with the path of *p* PCs – Intel benchmarks.

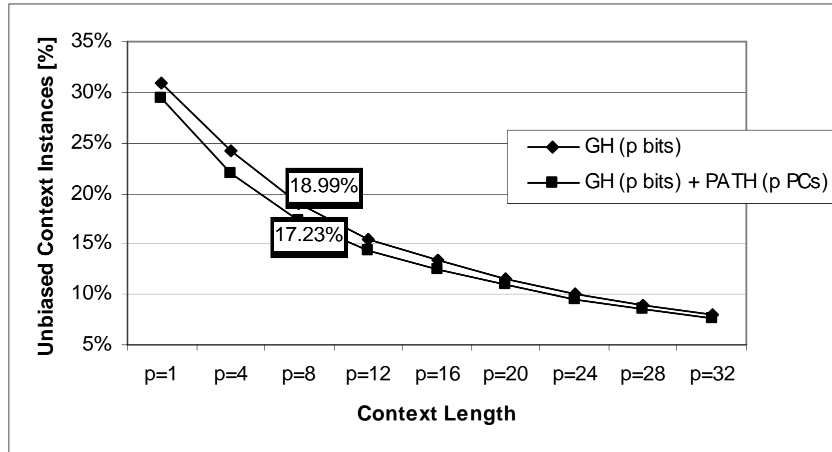

Figure 3.6. The gain introduced by the path for different context lengths – Intel benchmarks.

The main difference observed, analyzing the Figures 3.5 and 3.6, consists in the different values of these ratios (much bigger on SPEC benchmarks) – due to their different characteristics and functions [Loh05a]. However, it must mentioned that while SPEC benchmarks were simulated on 1 billion dynamic instructions the Intel benchmarks were entirely simulated, but the total number of dynamic instructions is lower (under 30 million).

Summarizing the statistics reported on the SPEC2000 benchmarks, 546 static branches generate 77,683,129 dynamic instances at average (142,120 instances / static branch). Focalizing now on those detected unbiased (with LH=28 bits, GH=28 bits, and GH XOR PC=28 bits), 113 static branches generate 4,376,664 dynamic instances at average (38,731 instances / static branch). Therefore the unbiased branches are generated by a few static branches having many dynamic instances. As a consequence, taking into account the enormous number of dynamic unbiased branches per a static branch, an adequate predictor has plenty of time to learn its behavior. The real problem is to find the right prediction information that changes such unbiased branches into biased ones.

3.2.3. An Analytical Model

High prediction accuracy is vital especially in the case of multiple instruction issue processors. Further, we assume the analytical model

proposed in [Cha94, Vin07], a superscalar processor that ignores stalls like cache misses and bus conflicts focalizing only about the penalty introduced by branch missprediction. Considering Branch Penalty (BP) as the average number of wasted cycles due to a branch missprediction for each dynamic instruction, it can be written the relation:

$$BP = C \cdot (1-Ap) \cdot b \cdot IR \quad [\textit{wasted clock / instruction}] \qquad (3.4)$$

Where we denoted:

C = number of penalty cycles wasted due to a branch missprediction;
Ap = prediction accuracy;
b = the ratio of branches (the number of branches reported to the total number of instructions);
IR = the average number of instructions that are executed per cycle (the superscalar factor of architecture; >1).

Following, we computed how many cycles take the execution of each instruction for a real superscalar processor that includes a branch predictor:

$$CPI_{real} = CPI_{ideal} + BP \quad [\textit{clock cycle / instruction}] \qquad (3.5)$$

Where:

CPI_{ideal} = represents the average number of cycles per instruction considering a perfect branch prediction ($Ap=100\% \Rightarrow BP=0$). It is obvious that $CPI_{ideal} < 1$.

CPI_{real} = represents the average number of cycles per instruction considering a real branch prediction ($Ap<100\% \Rightarrow BP>0 \Rightarrow CPI_{real} > CPI_{ideal}$).

Therefore, the real processing rate (the average number of instructions executed per cycle) results immediately from the following formula:

$$IR_{real} = \frac{1}{CPI_{real}} = \frac{1}{CPI_{ideal} + BP} \quad [\textit{instruction / clock cycle}] \qquad (3.6)$$

The relation (3.6) proves the non-linear correlation between processing rate (IR) and prediction accuracy (Ap). With these metrics, we adapted the model to our results obtained in Chapter 3. Further, we use the following notations:

x = the ratio of biased context instances;
$1 - x$ = the ratio of unbiased context instances.

In our simulations presented in [Gel06] we obtained using the *gshare* predictor [McFar93] the global prediction accuracy Ap_{global} = 93.60% (prediction applied to all branches) and respectively the accuracy of unbiased branch prediction $Ap_{unbiased}$ = 72.2% (only unbiased branches were predicted). Since Ap_{global} represents a weighted mean among predictions accuracies applied both to bias and unbiased branches, it can be determined the biased prediction accuracy Ap_{biased}.

$$Ap_{global} = x * Ap_{biased} + (1-x) * Ap_{unbiased} \qquad (3.7)$$

For previous example, $0.936 = 0.8253*A_{biased} + 0.1747*0.722$, resulting that $Ap_{biased} = 0.9813$.

Obviously, predicting the unbiased branches with a more powerful branch predictor having, to say, 95% prediction accuracy, determines a gain proportional with ratio of unbiased context instances: *Accuracy_gain* $=(0.95-0.722)*(1-x)$. More than that, this accuracy gain involves a processing rate speed-up according to (3.4) and (3.6). This gain justifies the importance and the necessity of finding and solving the difficult predictable branches. However, finding predictor that obtains so high prediction accuracy is beyond the scope of this paper.

Therefore, further we determined how much is influenced the branch penalty (BP) by the increasing of context length and what is the speed-up in these conditions. For this, we softly modified Chang's model [Cha94] by substituting Ap with our Ap_{global}, according to relation (3.7). Thus, the penalty introduced for missprediction of biased branches is the term $(1-Ap_{biased})*x$, respectively for considered wrong prediction of all unbiased branches ($Ap_{unbiased}$=0) is the term $(1-x)$.

Model proposed by Chang	**Our modified model**
$BP = C \cdot (1-Ap) \cdot b \cdot IR$	$BP = C \cdot b \cdot IR \cdot [1 - x \cdot Ap_{biased}]$ (3.8)

Figure 3.3 shows a decreasing of unbiased branches (1-*x*) by extending the context length that leads to a reduction of branch penalty (BP) according to (3.8), and implicitly to a greater IR according to (3.6). It can be written:

Context (Features Set) *Length* ↗ => *x* ↗ => BP ↘ => IR ↗ => ∃ Relative Speed-up>0.

Next, we computed the IR relative speed-up, varying the context length. Starting from the well known metric $\textit{Speed-up} = \frac{IR(L)}{IR(16)} \geq 1$, where L is the feature's length, $L \in \{20, 24, \text{and } 28\}$, we obtained the relative speed-up:

$$\textit{Relative Speed-up} = \frac{IR(L) - IR(16)}{IR(16)} \geq 0 \qquad (3.9)$$

Figure 3.7 illustrates the IR speed-up obtained extending the context. The baseline processor model has an IR_{ideal} of 4 [*instruction / clock cycle*] and incorporates a branch predictor with 98.13% prediction accuracy for biased branches. The considered number of penalty cycles wasted due to a branch missprediction in our model is 7. The ratio of simulated branches (the number of simulated branches reported to the total number of simulated instructions) is b=8% (see Table 3.1).

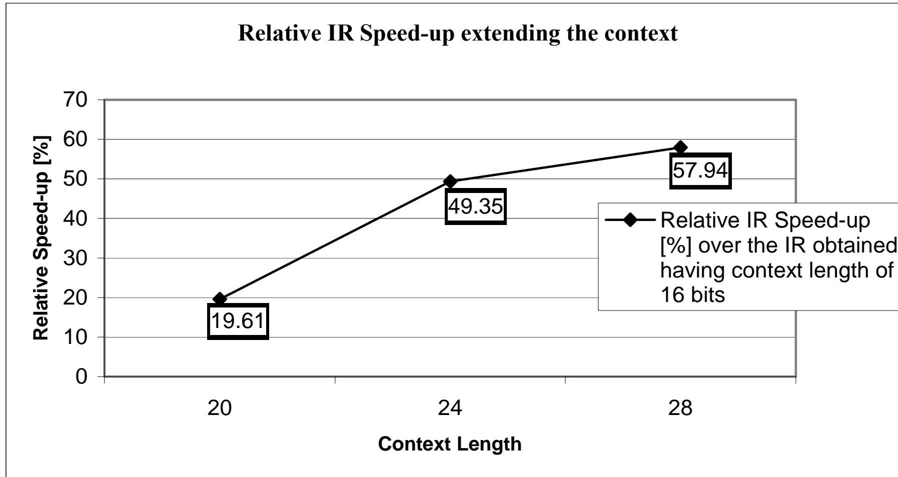

Figure 3.7. The IR relative speed-up obtained growing the context length.

Figure 3.7 illustrates not only the necessity of a greater number of prediction features to improve the processor performance, but also the necessity of new performing branch predictors that can consider a larger amount of information in making predictions (but whose size does not scale exponentially with the length of the input feature set).

3.2.4. An Example Regarding Branch Prediction Contexts Influence

In this section we analyze the contexts used by present day branch predictors (global and local histories respectively path information) from the point of view of their limits in predicting unbiased branches. The main idea is: in a perfect dynamic context all branch instances should have the same outcome. If the outcome is not the same a first solution might consists in extending the context information. After we varied the context length we observed that some dynamic contexts remained unpredictable despite of their length.

Related to the first part of our investigation – identifying the difficult-to-predict branches and quantifying them on testing programs, we used the traces obtained based on the eight C Stanford integer benchmarks, designed by Professor John Hennessy (Stanford University), to be computationally intensive and representative of non-numeric code while at the same time being compact. All these benchmarks were compiled by the *HSA gnu C* compiler, which targets the HSA (Hatfield Superscalar Architecture) instruction set. A dedicated HSA simulator [Ste97] that generates the corresponding traces simulated the resulted HSA object code. These helpful tools were developed at the University of Hertfordshire, Research Group of Computer Architecture, UK. The average instruction number is about 273.000 and the average percentage of branch instructions is about 18%, with about 76% of them being taken. Derived from HSA traces, special traces were obtained, containing exclusively all the processed branches. Each branch belonging to these modified HSA traces is stored in the following format: branch's type, the address of the branch (PC – program counter) and its target address. Some of these benchmarks are well known as very difficult to be predicted. For example, as Mudge et al. proved very clearly [Mud96], 75% accuracy could be an ultimate limit on *"quick-sort"* benchmark.

Following our aims, we developed an original dedicated trace-driven simulator that uses the above-mentioned traces [Rad07]. The most important input parameters for this simulator are the local/global history length (HRl

bits (l) / HRg bits (k)), number of entries in prediction table, the type of predictor, the simulated benchmark. As outputs, the simulator generates prediction accuracy, number of difficult-to-predict branches, and other useful statistics. Further, we present partially the C and assembly code of Stanford *Perm* benchmark that generates a suite of permutations. We detect unbiased branches and we focused on two of the most important branch instructions (having PC=35 and PC=58 after compiling process).

```
Permute (int n){
 int k;
 pctr = pctr+1;
 if(n != 1) {        # the first branch instruction analyzed (PC=35)
  Permute(n-1);
  for( k = n-1; k >= 1; k--){        # the second branch instruction analyzed (PC=58)
        Swap(&permarray[n], &permarray[k]);
        Permute(n-1);
        Swap(&permarray[n], &permarray[k]);
  };
 }
}


_Permute:
        SUB SP, SP, #128
        ST 0(SP), RA
        ST 8(SP), R17
        ST 12(SP), R18
        ST 16(SP), R19
        ST 20(SP), R20
        MOV R20, R5
        LD R13, _pctr
        ADD R13, R13, #1
        ST _pctr, R13
        EQ B1, R20, #1
        BT B1, L8 (#0)   # after compiling process this branch has the address 35
                     (PC=35)
        ADD R17, R20, #-1
        MOV R5, R17
        BSR RA, _Permute (#0)
        MOV R18, R17
        LES B1, R18, #0
        BT B1, L8 (#0)
        ASL R13, R20, #2
        MOV R7, #_permarray
        ADD R19, R13, R7
        ASL R13, R18, #2
        ADD R17, R13, R7
```

```
L12:
        MOV R5, R19
        MOV R6, R17
        BSR RA, _Swap (#0)
        ADD R5, R20, #-1
        BSR RA, _Permute (#0)
        MOV R5, R19
        MOV R6, R17
        BSR RA, _Swap (#0)
        ADD R17, R17, #-4
        ADD R18, R18, #-1
        GTS B1, R18, #0
        BT B1, L12 (#0)  # after compiling process this branch has the address 58
                    (PC=58)
```

In the following simulations [Flo07] the settled parameters are: *Path* = not selected, *Unbiased polarization degree* = 0.95, HRl and HRg being the local and global history. We define polarization index (*bias*) of a certain branch context as:

$$\text{bias} = \max\left(\frac{T}{T+NT}, \frac{NT}{T+NT}\right) \tag{3.9}$$

where T and NT represent number of "*taken*" respective "*not taken*" branch instances corresponding to that certain context.

1. Parameters: **HRl = not selected, HRg on 3 bits**, => Unbiased contexts: **25.0**[%]
From the unbiased branches list we selected just two branch instructions in two global contexts:
PC: 35 HRg: 101 T: 2520 NT: 1100 Bias: 0.696
PC: 58 HRg: 111 T: 1419 NT: 3620 Bias: 0.718

2. Parameters: **HRl = not selected, HRg on 4 bits**, => Unbiased contexts: **17.813**[%]
PC: 35 HRg: **0**101 T: 840 NT: 260 Bias: 0.763
PC: 35 HRg: **1**101 T: 1680 NT: 840 Bias: 0.667
PC: 58 HRg: **0**111 T: 1419 NT: 1100 Bias: 0.563
PC: 58 HRg: **1**111 T: 0 NT: 2520 Bias: 1.000 => ***The branch with the address PC: 58 in context HRg: 1111 became fully biased.*** Practically it doesn't appear in the unbiased branch list.

3. Parameters: **HRl on 1 bit, HRg on 4 bits,** => Unbiased contexts: **17.813**[%]
PC: 35 HRg: **0**101 HRl: **0** T: 840 NT: 260 Bias: 0.763
~~PC: 35 HRg: **0**101 HRl: **1** this context doesn't occur~~
PC: 35 HRg: **1**101 HRl: **0** T: 1680 NT: 840 Bias: 0.667
~~PC: 35 HRg: **1**101 HRl: **1** this context doesn't occur~~
PC: 58 HRg: **0**111 HRl: **0** T: 1419 NT: 1100 Bias: 0.563
~~PC: 58 HRg: **0**111 HRl: **1** this context doesn't occur~~

4. Parameters: **HRl on 2 bits, HRg on 4 bits**, => Unbiased contexts: **9.673**[%]
PC: 35 HRg: 0101 HRl: **0**0 T: 840 NT: 260 Bias: 0.763
~~PC: 35 HRg: 0101 HRl: **1**0 – this context doesn't occur~~
~~PC: 35 HRg: 1101 HRl: **0**0 – this context doesn't occur~~
PC: 35 HRg: 1101 HRl: **1**0 T: 1680 NT: 840 Bias: 0.667
PC: 58 HRg: **0**111 HRl: **0**0 T: 1419 NT: 260 Bias: 0.845
PC: 58 HRg: **0**111 HRl: **1**0 T: 0 NT: 840 Bias: 1.000=> ***The branch with the address PC: 58 in context HRg: 0111 and*** HRl: **1**0 ***became fully biased.*** Practically it doesn't appear in the unbiased branch list.
…

5. Parameters: **HRl on 2 bits, HRg on 7 bits,** => Unbiased contexts: **9.668**[%]
PC: 58 HRg: 1110111 HRl: **0**0 T: 1419 NT: 260 Bias: 0.845

6. Parameters: **HRl on 2 bits, HRg on 8 bits,** => Unbiased contexts: **8.134**[%]
PC: 58 HRg: **0**1110111 HRl: **0**0 T: 579 NT: 260 Bias: 0.690
PC: 58 HRg: **1**1110111 HRl: **0**0 T: 840 NT: 0 Bias: 1.000=> ***The branch with the address PC: 58 in context HRg: 11110111 and*** HRl: ***00 became fully biased.*** Practically it doesn't appear in the unbiased branch list.

Conclusion: As it can be observed, *increasing the context length, some branches in certain contexts became fully biased, but a great percentage still remains unbiased*.

Comparing the previous results it can be observed that as or richer the context became, as smaller the unbiased branches percentage became. From the 1st case to 2nd one, the unbiased branches percentages decrease with 7.187% and it can be observed how the two unbiased branches, in small contexts, are still unsolved. However, the branch with the address PC: 58 became fully biased in context HRg: 1111 decreasing the number of unbiased branches with 2520. Practically it does not appear in the unbiased branch list. In the 3rd case (adding one bit of local history) the unbiased branches percentage remains unchanged. In the 4th local history is set on 2 bits and much more contexts became biased (the unbiased branches percentage decreases with 8.14%). Although, there are some contexts that remain unbiased (see above: PC: 35 HRg: *x*101 HRl: *x*0 – where *x* could be 0 or 1).

Analyzing the code sequence it can be observed that to reach conditional branch 58, the previously 3 branches are every time Taken (return from *permute* function, call of *swap* function and return – not necessarily correlated with the branch 58). One reason for the larger percentage of unbiased branches refers to the fact that the branches within the global history length may not have correlation with the current branch, or the relevant history might be too far away. If the context would permit it could be seen a correlation between branches situated at a large distance in the dynamic instruction stream. Recurrence and function calls hide some branches that are really correlated with the analyzed one. Also, the local

correlation reduces the noise included in global history. Similar examples we found in *tower* benchmark that solves the *Hanoi towers* problem.

The insufficiency of global correlation information is remarked also in the case of programs or data structures, which produce a variable number of history bits as the data changes (data correlation). This occurs in the link lists or trees cases where the address of an element is tested (usually comparison with 0) and then a recurrent call of the same function is generated to test the next element in the tree (left or right sub-tree). The same situation does occure in the *hash table* cases having link lists to solve the collisions. A possible solution could be to use data values or structural information to keep the predictor more synchronized with data. We tried such an approach in [Gel07b].

4. Predicting Unbiased Branches

This section presents some important present-day branch predictors and, respectively, some proposed condition-history-based branch predictors, all of them being used to evaluate, in terms of prediction accuracy, the unbiased branches identified in [Gel06, Vin06].

4.1. The Perceptron-Based Branch Predictor

Jiménez and Lin [Jim01] proposed a two-level scheme that uses fast single-layer perceptrons instead of the commonly used two-bit saturating counters. The branch address is hashed to select the perceptron, which is used to generate a prediction based on global branch history. In [Jim02] the authors developed a perceptron-based predictor that uses both local and global branch history in the prediction process. Figure 4.1 presents the architecture of the perceptron-based branch predictor.

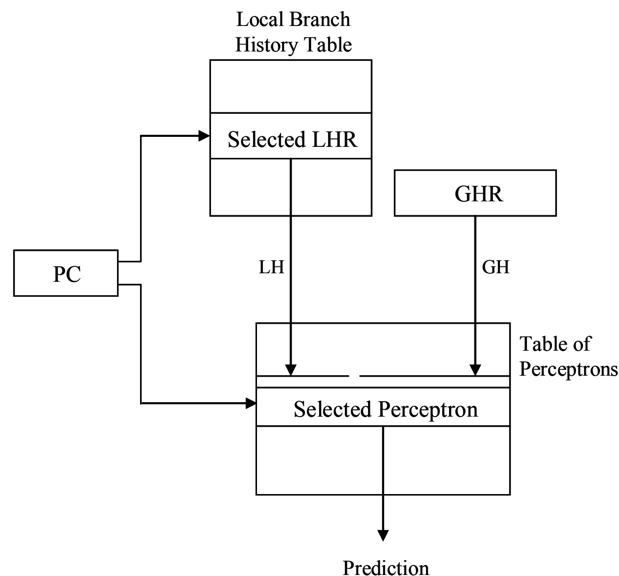

Figure 4.1. The perceptron-based branch predictor.

The lower part of the branch address (PC) selects a perceptron in the table of perceptrons (weights' matrix) and, respectively a local history register in the local branch history table. Both local and global branch history are used as inputs for the selected perceptron in order to generate a prediction.

4.2. The Idealized Piecewise Linear Branch Predictor

The piecewise linear branch prediction [Jim05], is a generalization of perceptron branch prediction [Jim01] and path-based neural branch prediction [Jim03]. The path-based neural predictor begins the branch's output computation in advance of the prediction, each computation step being processed as soon as a new element of the path is executed. Thus, the vector of weights used to generate prediction, is selected according to the path leading up to a branch – based on all branch addresses belonging to that path – rather than according to the current branch address alone as the original perceptron does. This selection mechanism improves significantly the prediction accuracy, because, due to the path information used in the prediction process, the predictor is able to exploit the correlation between the output of the branch being predicted and the path leading up to that branch. On the other hand, the prediction latency is almost completely hidden because the output's computation begins far in advance of the effective prediction. The most critical-timing operation is the sum of the bias weight and the current partial sum. To generate a prediction, the correlations of each component of the path are aggregated. This aggregation is a linear function of the correlations for that path. Since many paths are leading to a branch, there are many different linear functions for that branch, and they form a piecewise-linear surface separating paths that lead to predicted taken branches from paths that lead to predicted not taken branches. The piecewise linear branch prediction [Jim05], is a generalization of perceptron branch prediction [Jim01], which uses a single linear function for a given branch, and respectively path-based neural branch prediction [Jim03], which uses a single global piecewise-linear function to predict all branches. The piecewise linear branch predictors use a piecewise-linear function for a given branch, exploiting in this way different paths that lead to the same branch in order to predict – otherwise linearly inseparable – branches. The predictor has the same architecture as the perceptron-based branch predictor (see Figure 4.1). The weight selection mechanism of the idealized piecewise linear branch predictor is presented in Figure 4.2, where GH is the global history, PC is the branch's address and

GA is the path – an array of the addresses afferent to the last executed branches. Thus, the weight W_{bpg} corresponds to branch b ($1 \le b \le B$), its global history bit g ($1 \le g \le G$) and the pth PC ($1 \le p \le P$) from its path.

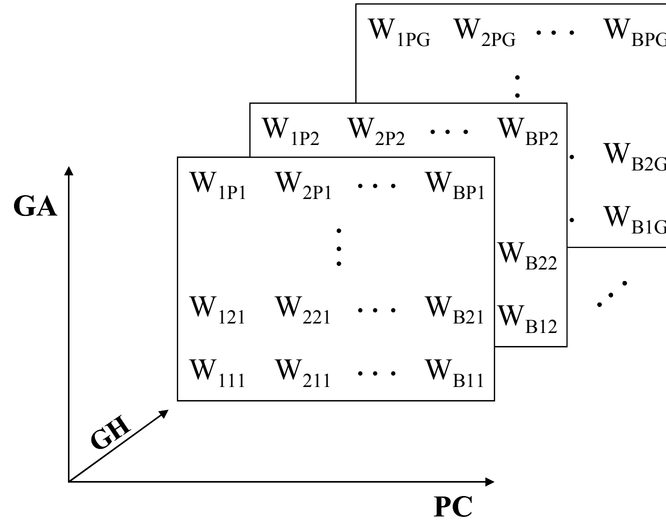

Figure 4.2. The weight selection mechanism of the idealized piecewise linear branch predictor.

For the Idealized Piecewise Linear Branch Predictor we used dynamically adjusted history lengths [Jim05]. The predictor counts the number of static branches whose bias magnitude, noted $|W_0|$, exceeds 2. If this number exceeds 300, then the predictor switches to lower global and local history lengths, otherwise, it uses higher global and local history lengths. This heuristic is applied after 300,000 branches have passed.

Related to Jiménez's research, we gave an original interpretation of his dynamically adjusting history length mechanism [Jim05], through our previously introduced "unbiased branches" concept [Gel06, Vin06]. Thus, his heuristics work as follows: if more than 300 "relatively biased" branches are encountered (branches having $|W_0|>2$), then it switches to lower global/local history length. Otherwise (meaning that there were encountered many "perfectly unbiased" branches, having $|W_0|\le 2$) it switches to higher global/local history length. From our point of view, this is justified by the fact that increasing history length reduces the number of unbiased branches.

4.3. The Frankenpredictor

The Frankenpredictor [Loh05a] is a *gskew-agree* global history predictor combined with a path-based neural predictor. The prediction mechanism of the Frankenpredictor is presented in Figure 4.3.

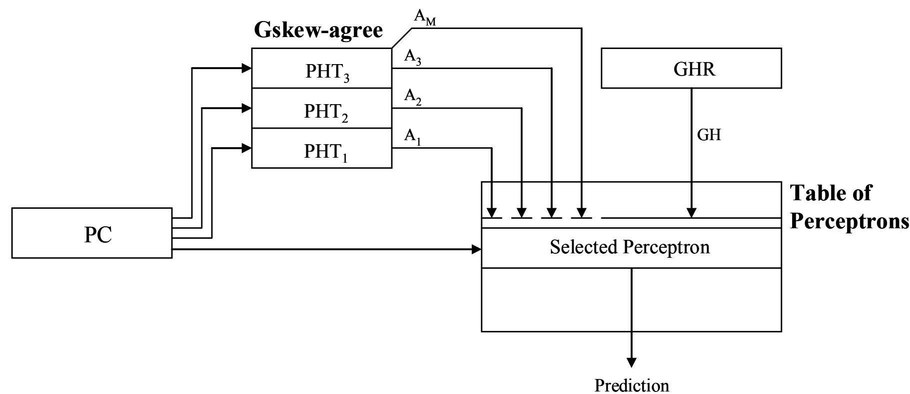

Figure 4.3. The Frankenpredictor's architecture.

The *gskew-agree* predictor avoids interference by mapping potential conflicting branches to different entries from three different tables. Three different predictions are provided, the final prediction being made by taking majority vote. The agreement approach uses a default BTFNT (backward taken forward not taken) static prediction (bias) for each branch. The predictions (P_1, P_2 and P_3) generated by the selected pattern history table entries are further compared with the bias. The neural predictor provides the ability of working with long branch histories and it also provides the hybridization by including the predictions of the *gskew-agree* predictor as additional bits in the perceptron's input vector – the agreement bits (A_1, A_2 and A_3) provided by the three PHTs (A_i is 1 if P_i agrees with the bias and 0 otherwise, $1 \leq i \leq 3$) and the majority vote (A_M).

4.4. The O-GEHL Predictor

The *Optimized GEometric History Length* (O-GEHL) predictor [Sez05] uses M distinct prediction tables indexed with hash functions of the branch address and the global branch history. Distinct history lengths of up

to 200 bits and a path history of up to 16 bits, consisting of 1 address bit per branch, are used to index the prediction tables. Table T_0 is indexed using the branch address. The history lengths used to index tables T_i, $1 \le i < M$, form a geometric series:

$$L(i) = \alpha^{i-1} \cdot L(1) \qquad (4.1)$$

The prediction tables store predictions as signed counters. To compute a prediction, a single counter is read from each prediction table. The prediction is computed as the sign of the sum S of the M counters. The prediction is taken if S is positive and not-taken otherwise. The prediction mechanism of the O-GEHL predictor is presented in Figure 4.4.

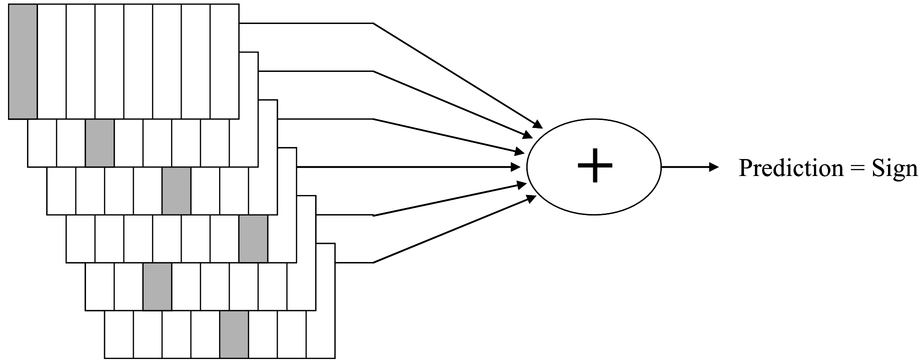

Figure 4.4. The O-GEHL predictor.

4.5. Value-History-Based Branch Prediction with Markov Models

The context-based predictor predicts the next value based on a particular stored pattern (context) that is repetitively generated in the value sequence. Theoretically they can predict any stochastic repetitive sequences. A context predictor is of order *k* if its context information includes the last *k* values, and, therefore, the search is done using this pattern of *k* values length. In fact, in this case the prediction process is based on a simple Markov model [Rab89].

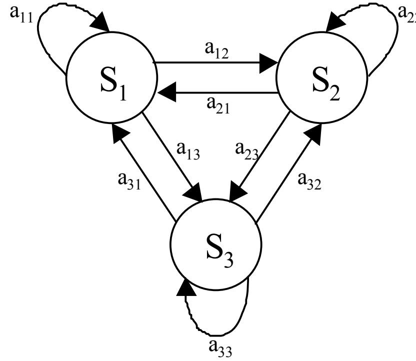

Figure 4.5. A Markov chain with 3 states.

A first order discrete Markov process may be described at any time as being in one of a set of N distinct states $S = \{S_1, S_2, ..., S_N\}$, as illustrated in Figure 4.5. A full probabilistic description of discrete Markov chain requires specification of the current state as well as all the predecessor states (the current state in a sequence depends on all the previous states). For the special case of a discrete, first order, Markov chain, this probabilistic description is truncated to just the current and predecessor state (the current state depends only on the previous state):

$$P[q_t = S_j | q_{t-1} = S_i,\ q_{t-2} = S_k,\ ...] = P[q_t = S_j | q_{t-1} = S_i] \quad (4.2)$$

where q_t is the state at time *t*. Thus, for a first order Markov chain with *N* states, the set of transition probabilities between states S_i and S_j is $A = \{a_{ij}\}$, where $a_{ij} = P[q_t = S_j | q_{t-1} = S_i]$, $1 \le i,\ j \le N$, having the properties $a_{ij} \ge 0$ and $\sum_{j=1}^{N} a_{ij} = 1$.

For a Markov chain of order R the probabilistic description is truncated to the current and R previous states (the current state depends on R previous states). The following example shows the necessity of using superior order Markov models. If the sequence of states is AAABCAAABCAAA, the Markov models of order 1 and respectively 2 mispredict A, and only a Markov Model of order 3 predicts correctly the next state B. This example is also presented in Figure 4.6.

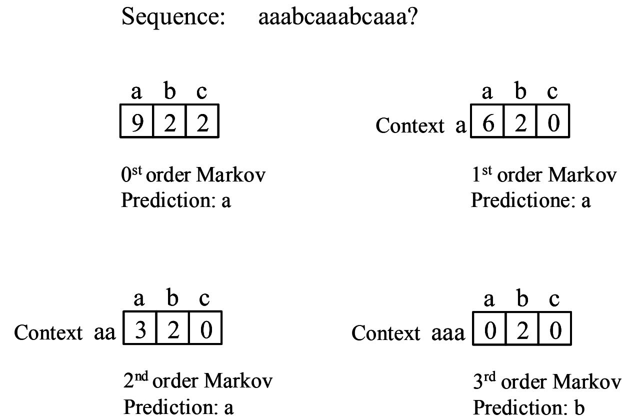

Figure 4.6. Markov predictors of different orders.

The predictors that implement the "*Prediction by Partial Matching*" algorithm (PPM) [Saz97] represent an important class of context-based predictors. Mudge et al. [Mud96] demonstrates that all two-level adaptive predictors implement special cases of the PPM algorithm that is widely used in data compression. It seems that PPM provides the ultimate predictability limit of two-level predictors. The PPM-based predictor contains a set of simple Markov predictors as it can be seen in Figure 4.6. It is predicted the value that followed the context with the highest frequency. In the case of complete-PPM predictor, if a prediction cannot be generated with the Markov predictor of order *k*, then the pattern length is shortened and the Markov predictor of order *k-1* tries to predict and so on.

4.5.1. Local Branch Difference Predictor

Figure 4.7 presents the speculative branch execution mechanism using a local PPM branch-difference predictor. The Branch Difference History Table (BDHT) maintains for each static branch the values or the signs of the inputs' differences (two approaches) corresponding to the branch's last *h* dynamic instances (B_1, B_2, ..., B_h). It would be possible to keep the differences corresponding to the previous *h* branches, therefore a global correlation approach instead of a local approach. Obviously, hybrid global-local approaches should be possible and useful too. Regarding the approach that uses only the signs of the input differences, a value of 1 in the history indicates that the corresponding branch difference was positive, a -1

indicates a negative difference, while a 0 indicates equality between the branch inputs. The BDHT entry is selected by the branch address (PC of B_0). The branch differences from the selected BDHT entry represent the PPM's input. Thus, the sign of the input difference (-1, 1, or 0) corresponding to the current branch (B_0) is predicted, using the complete-PPM algorithm of order k, where $k<h$ (see Figure 4.6). The branch B_0 is executed speculatively using the predicted inputs' difference only if the considered pattern of length k is repeated in the string of last h differences with a frequency greater or equal than a certain threshold.

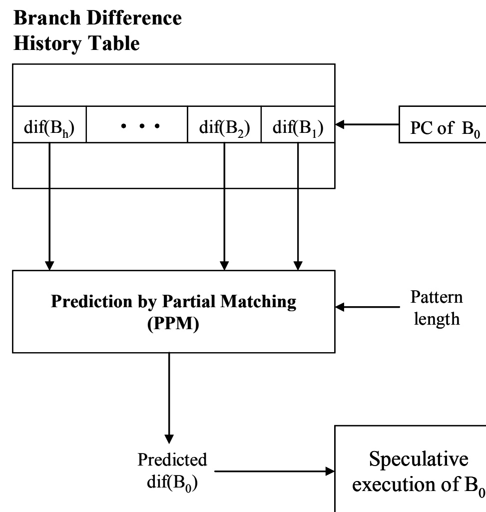

Figure 4.7. Speculative branch execution using local complete-PPM branch-difference predictors.

4.5.2. Combined Global and Local Branch Difference Predictor

Figure 4.8 presents the hybrid speculative branch execution mechanism using a combined global and local PPM-based branch-difference predictor. The Global History Register (GHR) contains the global history: the global branch difference history or the global branch outcome history (two different approaches). For each global history pattern, a distinct BDHT is maintained. Thus, the BDHT is selected by the GHR. A certain BDHT

contains for each static branch the inputs' differences corresponding to the branch's last *h* dynamic instances (B_1, B_2, ..., B_h). The selected BDHT is indexed by the branch address (PC of B_0). The branch differences from the selected BDHT entry represent the input for the PPM. Thus, the sign of the input difference (-1, 1, or 0) corresponding to the current branch (B_0) is predicted, using the complete-PPM algorithm of order k, where k<h (see Figure 4.6). The branch B_0 is executed speculatively using the predicted inputs' difference only if the considered pattern of length k is repeated in the string of last h differences with a frequency greater or equal than a certain threshold.

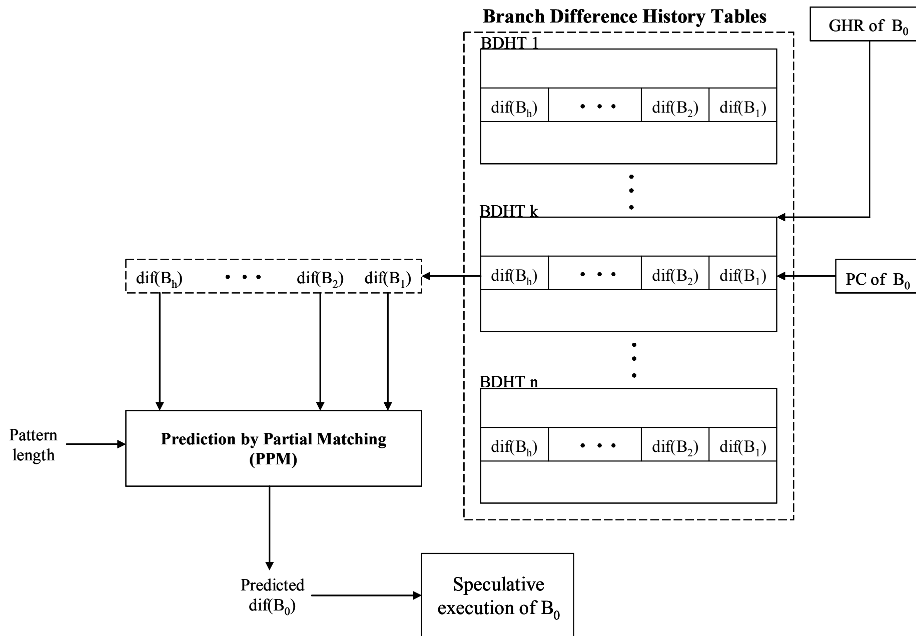

Figure 4.8. Speculative branch execution using global-local complete-PPM branch-difference predictors.

4.5.3. Branch Difference Prediction by Combining Multiple Partial Matches

Figure 4.9 presents the speculative branch execution mechanism using the *Branch-Difference Predicion by Combining Multiple Partial Matches* algorithm. The Branch Difference History Table (BDHT)

maintains for each static branch the signs of the inputs' differences (a value of 1 in the history indicates that the corresponding branch difference was positive, a -1 indicates a negative difference, and a 0 indicates equality between the branch's inputs) corresponding to the branch's last *h* dynamic instances (B_1, B_2, ..., B_h). A BDHT entry is selected by the branch's address (PC of B_0), as in the previous approaches. The branch differences from the selected BDHT entry represent the input for Markov predictors of different orders. Thus, the sign of the input difference (-1, 1, or 0) corresponding to the current branch (B_0) is predicted using multiple Markov predictors of orders ranging between [1, n], n<h (see Figure 4.9). The final branch difference prediction is generated through the majority vote.

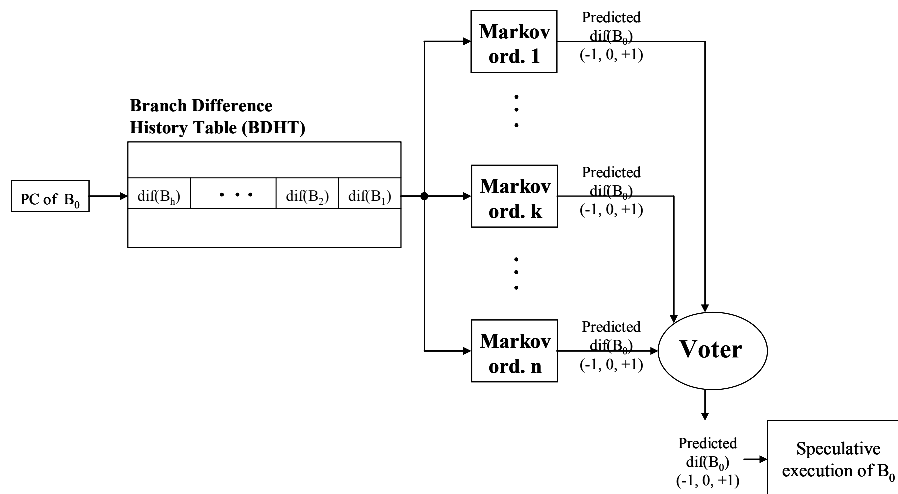

Figure 4.9. Speculative branch execution by combining multiple Markov branch-difference predictions.

Another possibility is to provide the final branch difference prediction through confidence-based voting. In this case, each BDHT entry maintains *n* saturated confidence counters associated to the *n* Markov predictors. The confidence counters ranging in our application between [-4, 4] are updated only if the corresponding Markov predictors provided a prediction (the pattern of length *k*, $1 \le k \le n$, was found at least once in the history of *h* values), by incrementing them in the case of a correct prediction and decrementing them otherwise. The confidence-based voting takes the majority, considering each Markov prediction as many times as the corresponding counter's value shows (only if this value is greater than zero).

We implemented and evaluated both these voting methods. Finally, the branch B_0 is executed speculatively using the predicted inputs' difference.

4.6. Experimental Results

The perceptron and our branch difference predictors were implemented by extending the *sim-bpred* simulator from *SimpleSim-3.0* [Sim]. We also implemented the unbiased branch selection mechanism and, thus, the predictors can be evaluated on unbiased branches too. We evaluate programs from the SPECcpu2000 benchmark suite, especially those that indicated a high percentage of unbiased branches [Gel06, Vin06]. The Championship Branch Prediction (CBP-1) simulators afferent to the Frankenpredictor [Loh05a] and respectively the Piecewise Linear Branch Predictor [Jim05] were extended to work with the same unbiased branch selection mechanism. In order to exploit these predictors we used the CBP-1 branch prediction framework which includes twenty traces (5 integer programs, 5 floating point, 5 multimedia applications and 5 server benchmarks) and a driver that reads the traces and calls the branch predictor [CBP04]. The traces are approximately 30 million instructions long and include both user and system codes. The two predictors were implemented within the constraints of a storage budget of (64K + 256) bits.

All simulation results are reported on 1 billion dynamic instructions skipping the first 300 million instructions from the SPEC2000 benchmarks [SPEC] and, respectively, on all instructions from the INTEL benchmarks [CBP04]. We note with LH(p)-GH(q) prediction information consisting in local history (LH) of *p* bits, and global history (GH) of *q* bits. We also note with *PPM(tdim, hlen, plen, thres, htype)* a complete-PPM branch-difference predictor using a Branch Difference History Table (BDHT) of *tdim* entries, a history length of *hlen* differences, a search pattern length of *plen* (specifying the current state), a threshold of *thres*, and considering a history of branch difference values or branch difference signs (*htype*=value/sign).

4.6.1. Evaluating Neural-Based Branch Predictors

In the first stage of this work, we'll measure with present-day branch predictors the prediction accuracy on all branches and, respectively, only on the final list of unbiased branches identified in [Vin06], using different local and global history lengths. Table 4.1 shows comparatively the results

obtained on the SPEC2000 benchmarks using a simple perceptron-based predictor integrated into *Simplesim-3.0* [Sim].

	LH(28)-GH(0)		LH(0)-GH(28)		LH(28)-GH(28)		LH(14)-GH(14)		LH(28)-GH(40)	
Bench	**All**	**Unb.**	**All**	**Unb.**	**All**	**Unb.**	**All**	**Unb.**	**All**	**Unb.**
bzip	87.3	70.1	90.7	74.8	90.6	74.8	90.5	74.8	90.6	74.7
gzip	85.7	77.9	91.5	79.1	91.9	79.3	91.6	79.3	92.1	79.9
mcf	87.3	51.0	98.5	69.4	98.7	72.5	98.3	67.5	98.8	73.7
parser	85.2	60.7	93.5	69.0	93.9	69.7	93.3	68.4	94.0	70.6
twolf	79.9	60.2	86.2	66.2	87.0	68.2	85.6	66.0	87.2	68.2
Mean	85.1	64.0	92.1	71.7	92.4	72.9	91.9	71.2	**92.5**	**73.4**

Table 4.1. The prediction accuracies obtained with the perceptron predictor using different prediction information on all branches and, respectively, only on unbiased branches from the SPEC2000 benchmarks. We used a table of perceptrons with 256 entries.

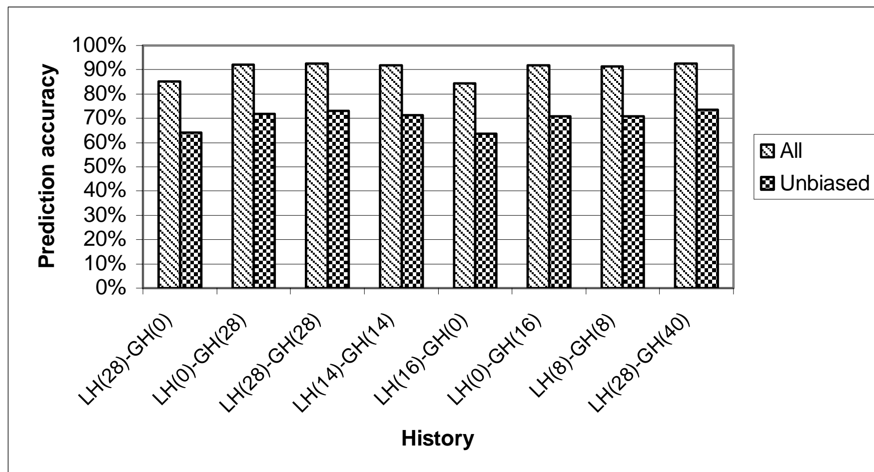

Figure 4.10. The average prediction accuracies obtained with the perceptron predictor using different prediction information on all branches and, respectively, only on unbiased branches from the SPEC2000 benchmarks. We used a table of perceptrons with 256 entries.

Table 4.1 intends to find an optimal LH(p)-GH(q) configuration within an enormous space of possible solutions. We did not use a well-known heuristic search method (e.g. genetic algorithms), preferring an empirical one based on our experience in the branch prediction field. As Table 4.1 and Figure 4.10 show, when we used the best configuration of the perceptron predictor (a local history of 28 bits and a global history of 40 bits –

determined based on laborious simulations), we obtained a prediction accuracy of 92.58% on all branches and, respectively, of only 73.46% on the unbiased branches.

Figures 4.11 and 4.12 show comparatively on the SPEC2000 benchmarks the prediction accuracies obtained with different present-day branch predictors on all branches and, respectively, only on the final list of unbiased branches identified in [Gel06, Oan06] using the XOR between the global history of 32 bits and the path of 32 PCs.

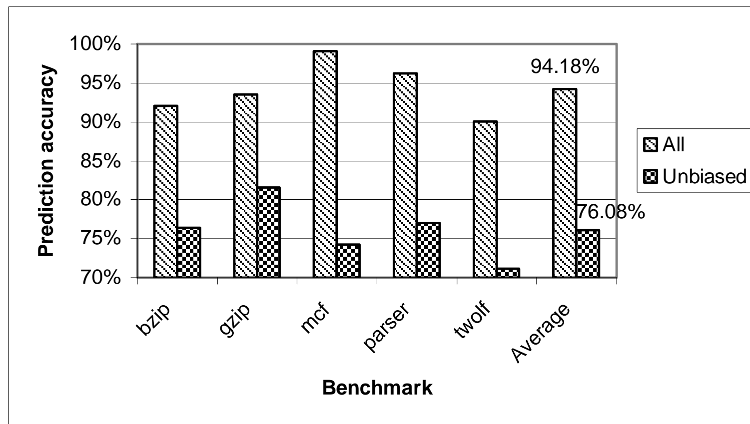

Figure 4.11. The average prediction accuracies obtained with the Frankenpredictor on the SPEC2000 benchmarks.

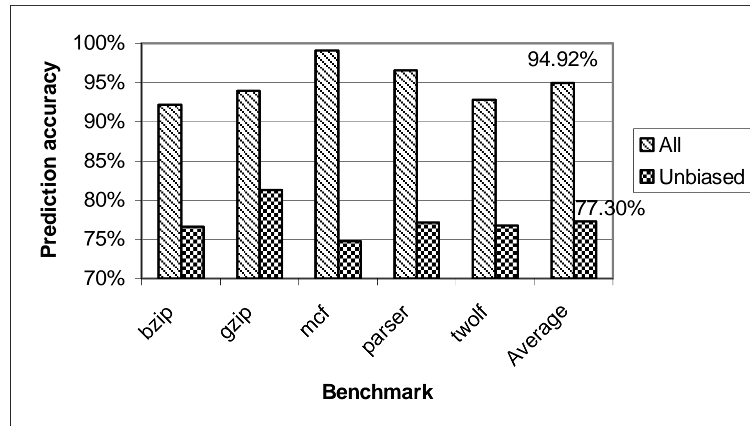

Figure 4.12. The average prediction accuracies obtained with the piecewise linear branch predictor on the SPEC2000 benchmarks.

We measured the prediction accuracies with the Frankenpredictor [Loh05a], and the Idealized Piecewise Linear Branch Predictor [Jim05], both described in the previous sections. We used the original Idealized Piecewise Linear Branch Predictor where the global history length is dynamically adjusted between 18 and 48 bits and, respectively, the local history length between 1 and 16 bits. For the Frankenpredictor we used a global history of 59 bits. Even if the Idealized Piecewise Linear Branch Predictor doesn't solve satisfactory the unbiased branches problem, it predicts them with an average accuracy of 77.3% that is better than all the other simulated branch prediction schemes.

	Frankenpredictor		**Piecewise**	
Benchmark	**All**	**Unb.**	**All**	**Unb.**
dist-fp-1	98.5	71.9	98.4%	78.2
dist-fp-2	99.1	95.4	99.0%	97.5
dist-fp-3	99.6	96.0	99.6%	99.1
dist-fp-4	99.9	90.3	99.8%	95.2
dist-fp-5	99.9	84.7	99.8%	96.8
dist-int-1	97.6	79.9	98.3%	87.9
dist-int-2	93.4	81.5	94.0%	85.7
dist-int-3	91.3	71.7	93.2%	79.2
dist-int-4	98.9	91.4	98.6%	92.1
dist-int-5	99.7	74.1	99.7%	88.0
dist-mm-1	92.8	83.8	93.0%	85.7
dist-mm-2	90.6	84.6	91.0%	89.3
dist-mm-3	99.1	67.9	99.4%	87.0
dist-mm-4	98.6	98.7	98.6%	98.9
dist-mm-5	95.2	85.4	95.2%	88.8
dist-serv-1	97.8	83.5	97.5%	89.4
dist-serv-2	97.7	83.7	97.6%	89.2
dist-serv-3	95.6	84.8	95.1%	88.9
dist-serv-4	96.3	77.5	96.4%	83.2
dist-serv-5	96.7	75.2	96.7%	82.2
Average	96.9	83.1	97.0%	89.1

Table 4.2. The prediction accuracies obtained with the piecewise linear branch predictor and the Frankenpredictor on the Intel benchmarks.

Table 4.2 and Figures 4.13 and 4.14 show comparatively on the CBP-1 Intel benchmarks [CBP04] the prediction accuracies obtained on all branches and, respectively, only on the final list of unbiased branches identified in [Gel06, Oan06] using the XOR between the global history of 32 bits and the path of 32 PCs. We measured the prediction accuracies on the Intel benchmarks with the Idealized Piecewise Linear Branch Predictor [Jim05]

and the Frankenpredictor [Loh05a]. We used for both predictors the same configurations as on the SPEC2000 benchmarks. Even if the Idealized Piecewise Linear Branch Predictor doesn't solve satisfactory the unbiased branches problem, it predicts them with an average accuracy of 89.1% that is better than all the other simulated branch prediction schemes. However, we are reserved regarding the CBP-1 Intel benchmarks due to their shortness. Furthermore, the *Second Championship Branch Prediction Competition* (CBP-2) [CBP06] have used all the twelve CPUintSPEC2000 benchmarks and eight JavaSPECjvm98 benchmarks.

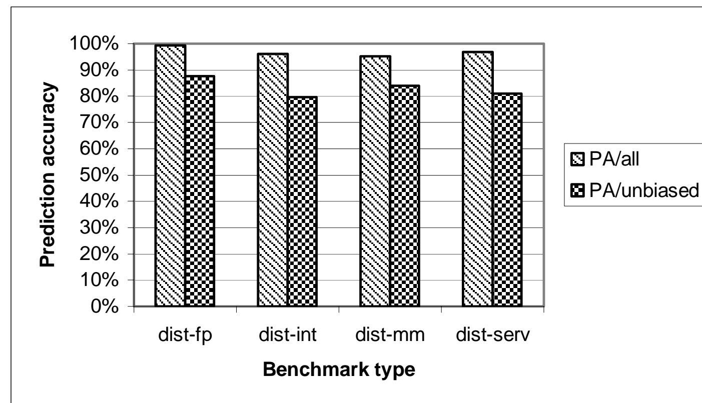

Figure 4.13. The average prediction accuracies obtained with the Frankenpredictor on the Intel benchmarks.

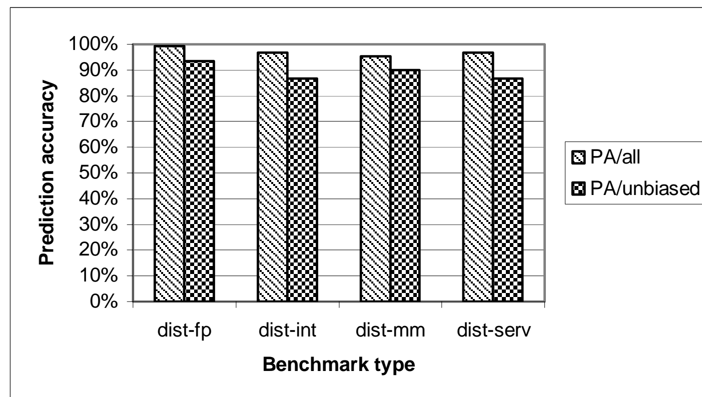

Figure 4.14. The average prediction accuracies obtained with the piecewise linear branch predictor on the Intel benchmarks.

We empirically found out that the behavior of difficult branches – as we defined them – cannot be sufficiently learned neither by neural predictors. Figures 4.11, 4.12, 4.13 and 4.14 confirm us again, that the unbiased branches, identified in our previous work [Vin06, Gel06], are hard-to-predict with present-day branch predictors.

4.6.2. Evaluating the O-GEHL Predictor

We have also evaluated the Optimized GEometric History Length (O-GEHL) predictor [Sez05], described in section 4.4 (see Figure 4.4). We used an 8-table O-GEHL predictor. The experimental results obtained on the SPEC2000 benchmarks are presented in Figure 4.15.

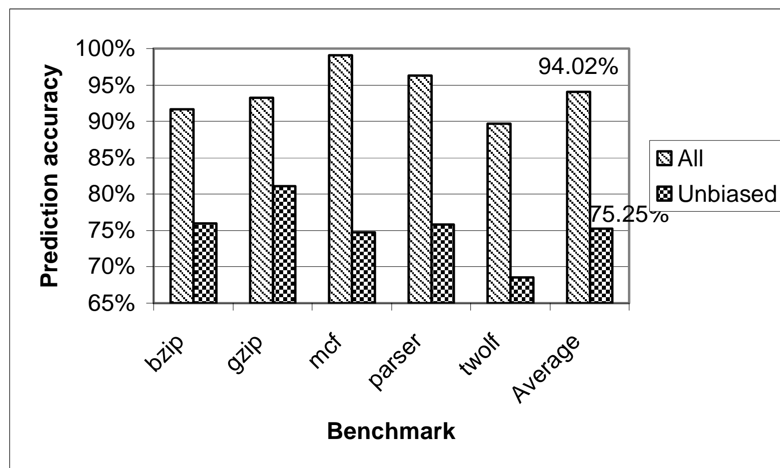

Figure 4.15. The average prediction accuracies obtained with the O-GEHL predictor on the SPEC2000 benchmarks.

As it can be observed, the neural branch predictors provided higher prediction accuracy then the O-GEHL predictor (see comparatively Figures 4.11, 4.12 and 4.15).

4.6.3. Evaluating Local Branch Difference Predictors

We'll continue this work by evaluating the prediction accuracy of the complete-PPM branch-difference predictor (see Figure 4.7) on all

branches and, respectively, only on the final list of unbiased branches (identified in [Vin06]). We started our simulations by evaluating different local history lengths. Table 4.3 shows comparatively the results obtained on the SPEC2000 benchmarks, using a history of branch difference values and, respectively, a history of branch difference signs (-1 if negative, 1 if positive, or 0), considering an unlimited BDHT, a pattern length of 3, and a threshold of 1.

	History of Branch Difference Values		**History of Branch Difference Signs**	
History	**All**	**Unb.**	**All**	**Unb.**
LH(8)	85.78%	64.76%	86.56%	65.33%
LH(16)	86.84%	66.35%	88.34%	68.26%
LH(24)	86.79%	66.52%	88.66%	68.61%
LH(32)	86.83%	66.87%	88.88%	68.78%
LH(40)	86.81%	66.91%	89.03%	68.98%
LH(48)	86.77%	67.04%	89.11%	69.12%
LH(56)	86.78%	67.33%	89.19%	69.23%
LH(64)	86.76%	67.43%	89.26%	69.37%
LH(128)	86.56%	67.52%	89.45%	69.70%
LH(256)	86.39%	67.94%	89.58%	69.75%

Table 4.3. The average prediction accuracies on all branches and, respectively, only on unbiased branches from the SPEC2000 benchmarks, using branch-difference predictors with different local history lengths.

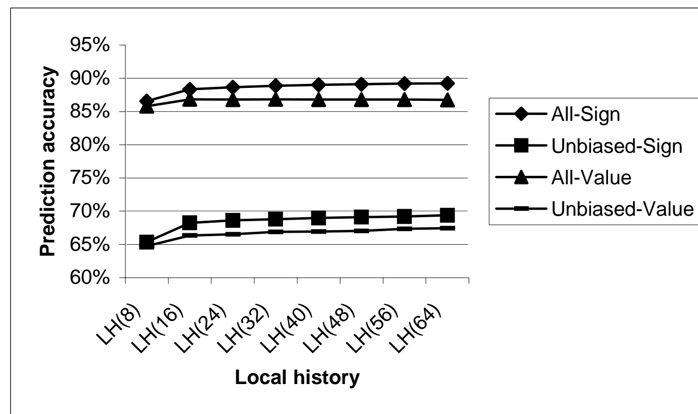

Figure 4.16. The average prediction accuracies on the SPEC2000 benchmarks, using a *PPM(tdim=unlimited, hlen=varied, plen=3, thres=1, htype=value and sign)* branch difference predictor with different local history lengths.

Figure 4.16 shows the difference's value prediction accuracies obtained on the SPEC2000 benchmarks, using an unlimited BDHT containing the values respectively the signs of the last branch differences, a pattern length of 3, and a threshold of 1. As simulations show (Figure 4.16), branch differences can be better predicted when only difference signs are used as history instead of difference values. Consequently, the sign of the current branch difference is better correlated with the signs of its previous differences than with the values of those differences.

The experimental results also show that the performance is relatively saturated starting with a local history length of 24 bits. Why is better to use only the signs of differences as history information instead of the values of differences? The number of distinct symbols that can occur in a value history is huge reported to only three symbols that can appear in a sign history. Thus, the frequency of symbols in a value history is very low. In the following example only a Markov predictor of order 1 can be used for the value history, and it generates a misprediction, while in the case of the sign history, even a Markov predictor of order 5 can be used, which generates the correct prediction:

Value history: -126, -34, 7, -42, -28, 75, -829, -7982, 102, -542, -42, ?
Sign history: -1, -1, 1, -1, -1, 1, -1, -1, 1, -1, -1, ?

Obviously, through a sign history much deeper correlations can be exploited than with a value history.

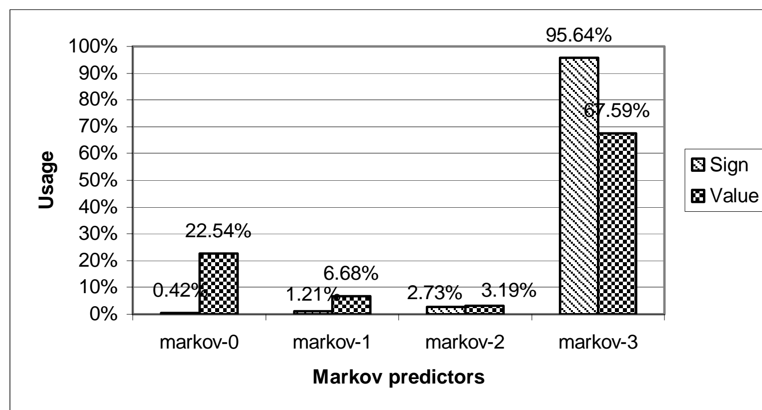

Figure 4.17. The average usage rates of Markov predictors using *PPM(tdim=unlimited, hlen=24, plen=3, thres=1, htype=sign and value)* branch difference predictors on all branches.

Figure 4.17 compares the sign history with the value history in terms of usage rate afferent to Markov predictors of different orders. We used the optimal history length 24 and a pattern length of 3, and therefore, we evaluated the usage rates corresponding to Markov predictors of orders 0, 1, 2 and 3. As Figure 4.17 shows, more often are used superior order Markov predictors by using a sign history, and thus, deeper correlations can be exploited. Therefore, we continued by evaluating different pattern lengths using an unlimited BDHT, a sign history of 24 branch difference signs, and a threshold of 1. As Figure 4.18 shows, the best PPM's pattern length is 3, considering the optimal local history of 24 branch difference signs.

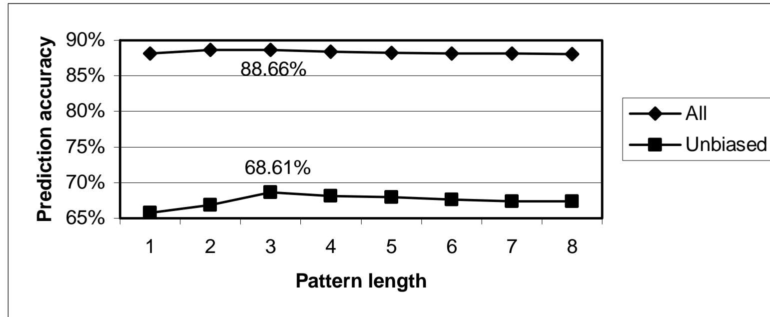

Figure 4.18. The average prediction accuracies on all branches and, respectively, only on unbiased branches from the SPEC2000 benchmarks, using a *PPM(tdim=unlimited, hlen=24, plen=varied, thres=1, htype=sign)* branch difference predictor with different pattern lengths.

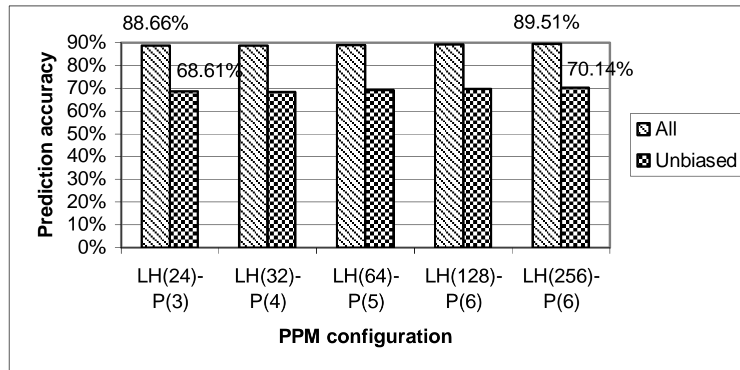

Figure 4.19. The average prediction accuracies on SPEC2000 benchmarks using a *PPM(tdim=unlimited, hlen=varied, plen=varied, thres=1, htype=sign)* branch difference predictor exploring different local history lengths and pattern lengths.

Figure 4.19 explores the space of local history lengths and pattern lengths using a threshold of 1 and confirms that an acceptable choice (taking into account a good accuracy/complexity trade-off report) is to use a history of 24 branch difference signs with a pattern length of 3.

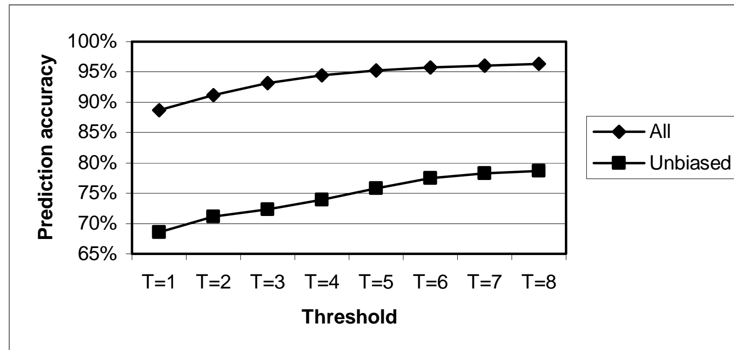

Figure 4.20. The average confidence on all branches and, respectively, only on unbiased branches from the SPEC2000 benchmarks, using a *PPM(tdim=unlimited, hlen=24, plen=3, thres=varied, htype=sign)* branch difference predictor with different threshold values.

Threshold	Lost predictions [%]
T=1	0.00
T=2	7.59
T=3	13.37
T=4	17.31
T=5	20.50
T=6	23.40
T=7	25.13
T=8	26.98

Table 4.4. Average percentages of predictions lost with different thresholds.

We also studied the influence of the threshold's value over the prediction accuracy, using an unlimited BDHT, a local history of 24 branch difference signs, and a pattern length of 3. The threshold's value means how many times the current search pattern must be found in the history string in order to generate a prediction, implementing thus a confidence degree (otherwise, no prediction is generated). Strictly considering the confidence metric, the experimental results presented in Figure 4.20 show that the optimal threshold value is 7. However, in this case, the total number of predictions decreases at average with 25.13% (see Table 4.4). Considering T=1, the

global prediction accuracy on unbiased branches A(T=1) is 68.61%. In contrast, considering T=7, the global accuracy A(T=7) is 74.87%x78.33% = 58.64% and, respectively, for T=2, A(T=2) is 92.41%x71.16% = 65.75%. Therefore, from the global accuracy point of view T=1 is optimal. The last parameter we varied is the dimension of the BDHT.

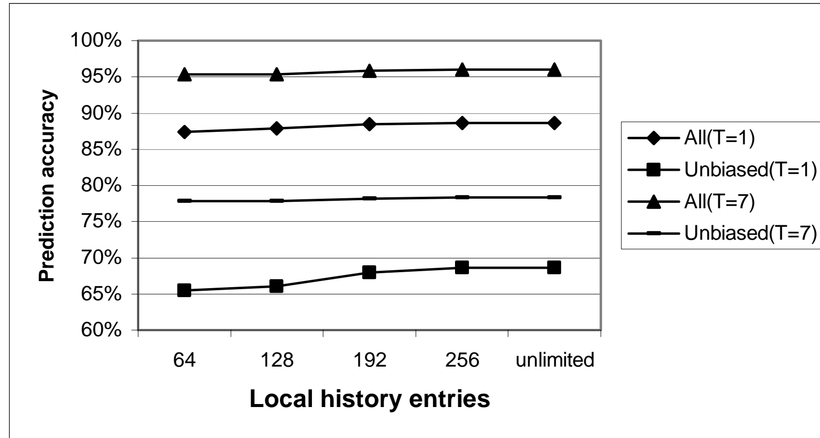

Figure 4.21. The average prediction accuracies on the SPEC2000 benchmarks using a *PPM(tdim=varied, hlen=24, plen=3, thres=1 and 7, htype=sign)* branch difference predictor considering different BDHT dimensions.

Figure 4.21 shows that a BDHT with 256 entries provides the same results as an unlimited BDHT does. Consequently, we determined that the optimal branch difference predictor configuration is *PPM(tdim=256, hlen=24, plen=3, thres=1 or 7, htype=sign*). The signs of branch differences can be predicted considering this optimal configuration with an accuracy of 68.60% on the unbiased branches and 88.66% on all branches and, respectively, a confidence of 78.33% on the unbiased branches and 96.05% on all branches.

The next step consists in speculatively executing branches based on their predicted input differences. The final confidence branch prediction accuracies – evaluating all branches and, respectively, only unbiased branches –, obtained using the speculative branch differences generated with the optimal branch difference predictor, are presented in Tables 4.5 (without threshold) and 27 (with threshold).

The average prediction accuracy obtained without threshold on the unbiased branches is only 71.76% (see Table 4.5). Using a threshold of 7, it grows to 79.69% (see Table 4.6).

	Branch Prediction Accuracy [%]	
Benchmark	**All**	**Unb.**
bzip	89.92	74.50
gzip	88.95	79.06
mcf	97.10	66.25
parser	91.47	66.01
twolf	85.29	73.00
Average	**90.55**	**71.76**

Table 4.5. The final branch prediction accuracies on all branches and, respectively, only on unbiased branches, obtained by using the speculative branch differences generated with the optimal branch-difference predictor without threshold.

	Branch Prediction Accuracy [%]	
Benchmark	**All**	**Unb.**
bzip	96.88	79.94
gzip	95.99	86.28
mcf	99.19	75.14
parser	96.71	73.26
twolf	93.40	83.83
Average	**96.43**	**79.69**

Table 4.6. The final branch prediction accuracies on all branches and, respectively, only on unbiased branches, obtained by using the speculative branch differences generated with the optimal branch-difference predictor using a threshold of 7.

The average prediction accuracy measured only on unbiased branches and, respectively, on all branches is lower for the complete-PPM predictor comparing with the perceptron predictor. Consequently, unbiased branches remain hard-to-predict even with the sign of the condition's difference in the local approach, due to the quasi-random values afferent to the branch condition. Therefore, a hybrid global-local approach is necessary.

4.6.4. Evaluating Combined Global and Local Branch Difference Predictors

In the combined global and local approach, each global history pattern points to its own BDHT (see Figure 4.8). The selected BDHT is indexed by the PC, as in the local approach. First, we evaluated the predictor by maintaining in the GHR (see Figure 4.8) the global branch

difference history: the signs of the inputs' differences corresponding to the previous *h* branches. Figure 4.22 shows comparatively the results obtained with and without threshold on all branches and, respectively, only on the unbiased branches from the SPEC 2000 benchmarks.

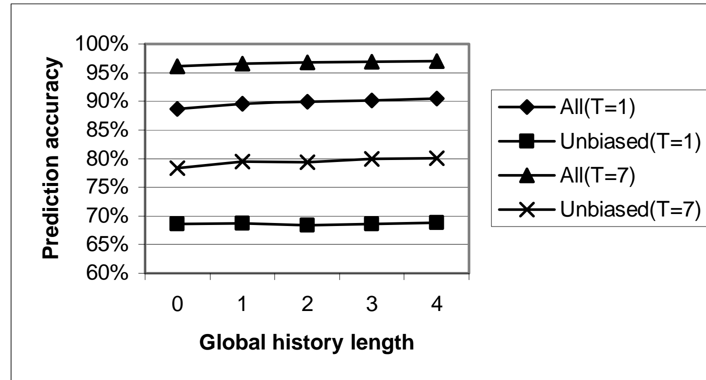

Figure 4.22. The average confidence on the SPEC2000 benchmarks using a *PPM(tdim=256, hlen=24, plen=3, thres=1 and 7, htype=sign)* branch difference predictor considering different global branch difference history lengths.

We also evaluated the predictor by maintaining in the GHR the global branch outcome history (Taken / Not Taken). Our simulation results show that the confidence is slightly better on unbiased branches if we use the global difference-sign history. Considering a global history length of 4 (GH=4), we obtained a confidence of 68.81% with the global difference-sign history, opposite to 67.84% obtained with global branch outcome history. The difference-sign history can be more efficient because, due to its additional information, it can efficiently exploit shorter contexts, too. The following example presents the situation for *bgez*:

Difference history: 138, 52, 47, 0, -591, 5783, 4, 702, 0, -35, 721, 5, 14, 0, ?
Sign history: +, +, +, 0, -, +, +, +, 0, -, +, +, +, 0, ?
Output history: T, T, T, T, NT, T, T, T, T, NT, T, T, T, T, ?

If after "0" statistically follows "-" (and, in the case of bgez, "0" is associated together with "+" to *Taken*) a first order Markov can correctly predict in the case of sign history, while, in the case of outcome history, the Markov predictor must be of order 4 or higher for correct prediction.

The signs of branch differences can be predicted, considering a *PPM(tdim=256, hlen=24, plen=3, thres=1, htype=sign)* having a global branch difference history of 4, with an accuracy of 68.81% on the unbiased branches and, respectively 90.47% on all branches (see Figure 4.22). The next step consists in executing branches based on their predicted input differences. The final branch prediction accuracies – evaluating all branches and, respectively, only unbiased branches –, obtained by using the speculative branch differences generated with this global-local branch difference predictor, are presented in Table 4.7. The results show that even the global-local PPM cannot improve the branch prediction accuracy obtained with the perceptron predictor.

	Branch Prediction Accuracy [%]	
Benchmark	**All**	**Unb.**
bzip	92.32	75.69
gzip	90.59	78.33
mcf	98.22	64.24
parser	93.90	69.14
twolf	86.62	70.28
Average	**92.33**	**71.54**

Table 4.7. The final branch prediction accuracies on all branches respectively only on unbiased branches, obtained using the speculative branch differences generated with the optimal global-local branch-difference predictor without threshold.

	Branch Prediction Accuracy [%]	
Benchmark	**All**	**Unb.**
bzip	97.62	84.44
gzip	96.70	86.36
mcf	99.53	74.46
parser	98.07	78.56
twolf	95.26	82.41
Average	**97.44**	**81.25**

Table 4.8. The final branch prediction accuracies obtained by using the optimal global-local branch-difference predictor with a threshold of 7 (confidence).

The final branch prediction accuracies – evaluating all branches and, respectively, only unbiased branches –, obtained by using the speculative branch differences generated using this global-local branch difference predictor with a threshold of 7, are presented in Table 4.8. As it can be observed, the global-local approach improves significantly the average prediction accuracy on all branches to 97.44%, if a threshold of 7 is used.

However, the average prediction accuracy remains still low on unbiased branches: 81.25%.

4.6.5. Branch Difference Prediction by Combining Multiple Partial Matches

Branch differences are predicted by five Markov predictors of orders ranging between [1, 5]. The final prediction is provided through majority voting, as we already presented in paragraph 4.5.3. We started our evaluations using a BDHT of 256 entries, local branch difference history of 24 values. Figure 4.23 presents the results obtained on the SPEC2000 benchmarks considering simple voting respectively confidence-based voting.

It can be observed that through confidence-based voting the branch differences can be predicted with a slightly higher accuracy than through simple voting.

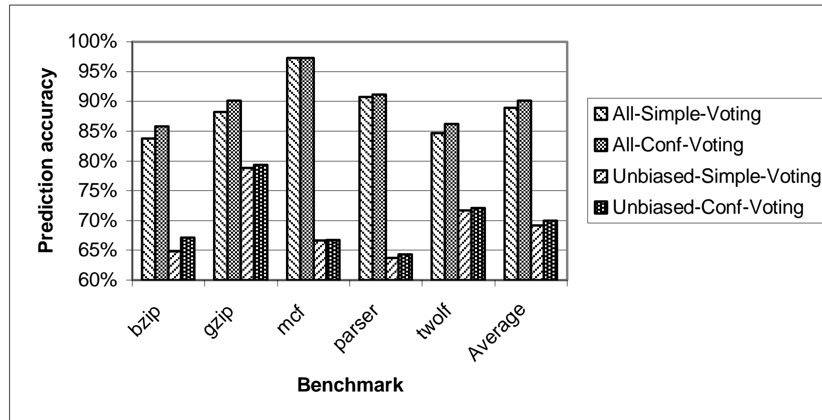

Figure 4.23. Branch difference prediction accuracies by combining multiple partial matches through simple voting and confidence-based voting.

Table 4.9 presents the final branch prediction accuracies – evaluating all branches and, respectively, only unbiased branches – obtained using the speculative branch differences generated by combining multiple partial matches through confidence-based voting.

	Branch Prediction Accuracy [%]	
Benchmark	**All**	**Unb.**
bzip	91.52	75.54
gzip	90.28	79.50
mcf	97.32	66.75
parser	92.27	67.13
twolf	86.55	72.30
Average	**91.59**	**72.24**

Table 4.9. The final branch prediction accuracies on all branches and, respectively, only on unbiased branches, obtained using the speculative branch differences generated by combining multiple partial matches through confidence-based voting.

Figure 4.24 shows again, that the unbiased branches identified in [Gel06, Oan06, Vin06] cannot be accurately predicted even with condition-history-based Markov predictors. The highest average prediction accuracy on the unbiased branches, of 77.30%, was provided by the piecewise linear branch predictor.

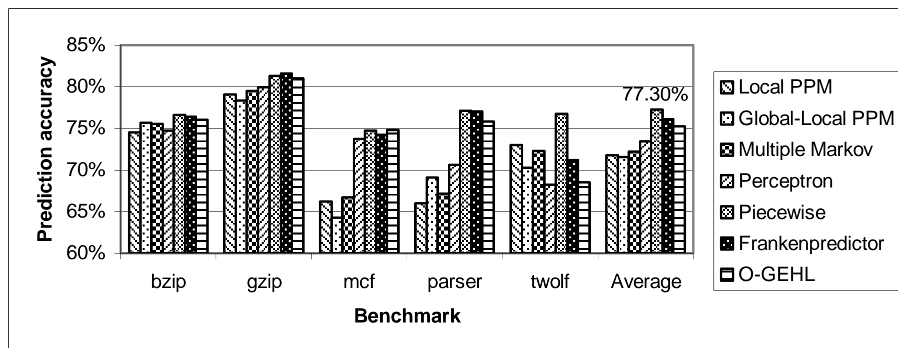

Figure 4.24. The final branch prediction accuracies obtained without threshold using the perceptron-based predictors, the O-GEHL predictor, the local complete-PPM, the global-local complete-PPM and respectively prediction by combining multiple partial matches through confidence-based voting, only on unbiased branches.

We also studied the influence of the threshold's value over the prediction accuracy by combining multiple partial matches through confidence-based voting, using a BDHT with 256 entries, and a local history of 24 branch difference signs. In this case, the confidence-based voting takes the majority, considering only Markov predictions found in the history string after the considered pattern at least T (threshold) times.

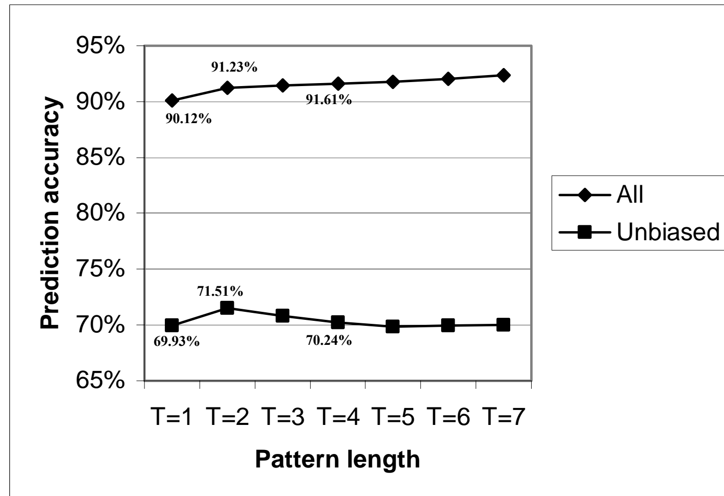

Figure 4.25. Branch difference prediction accuracies by combining multiple partial matches through confidence-based voting with different thresholds.

Threshold	Lost predictions [%]
T=1	2,25
T=2	5,20
T=3	6,62
T=4	8,06
T=5	9,40
T=6	10,78
T=7	13,02
T=8	2,25

Table 4.10. Average percentages of predictions lost by using different thresholds.

The experimental results presented in Figure 4.25 and Table 4.10 show that the optimal threshold value is 2. Thus, the final branch prediction accuracy by combining multiple partial matches through confidence-based voting with a threshold of 2 is 73.05% on unbiased branches.

5. Using Last Branch Difference as Prediction Information

Further, we evaluated the percentage of unbiased context instances using the last known branch condition difference together with global histories of *p* bits (1≤*p*≤24). A branch condition difference consists in the difference of the operand values implied in the branch condition. More than two branch condition differences are not necessary [Smi98, Hei99b]. Table 5.1 and Figure 5.1 compares the percentages of unbiased branches using the global history (GH), the global history concatenated with the path (GH + PATH), respectively the global history concatenated with the last branch difference (GH + LBD).

Context	p=1	p=4	p=8	p=12	p=16	p=20	p=24
GH (p bits)	49.28	39.38	32.08	27.23	24.46	22.08	20.23
GH (p bits) + PATH (p PCs)	42.19	36.39	29.71	25.51	23.13	20.74	19.01
GH (p bits) + LBD	36.99	32.25	26.94	22.39	19.91	17.85	**16.24**

Table 5.1. The gain introduced by the path respectively last branch difference (LBD) for different context lengths – SPEC2000 benchmarks [%].

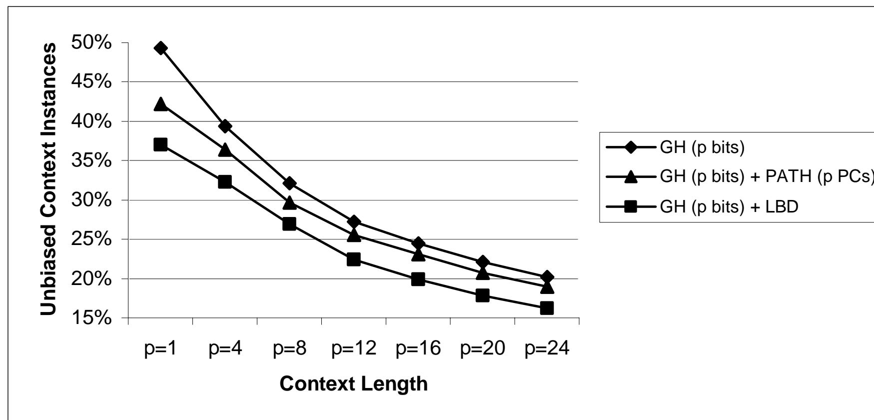

Figure 5.1. The gain introduced by the path respectively last branch difference (LBD) for different context lengths – SPEC2000 benchmarks.

The results, presented in Figure 5.1, show that the last branch condition is more efficient than the path information: it decreased the percentage of unbiased branches for all evaluated context lengths ($1 \leq p \leq 24$). Therefore we can use this new prediction information in some state-of-the-art branch predictors in order to increase prediction accuracy [Gel07a, Gel07b].

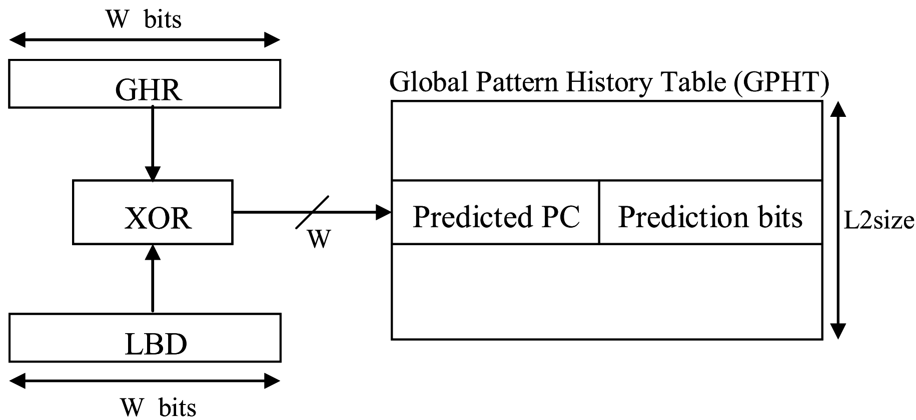

Figure 5.2. The GAg predictor using the last branch difference (LBD).

We first analyzed a GAg scheme that uses the last branch difference (LBD) by XORing it with the GHR (as the Gshare XORed the PC with the GHR). The predictor is presented in Figure 5.2. Table 5.2 presents the prediction accuracies obtained with the modified GAg predictor on unbiased branches.

Bench	GHPC16 (gshare)	GHLBD16	LBD4-GHLBD12	LBD8-GHLBD8	Shifted-GHLBD16	Shifted-LBD4-GHLBD12	Shifted-LBD8-GHLBD8	LBD4-GH12	Signed-LBD4-GHLBD12
bzip	67.40	66.16	69.66	70.26	66.55	69.45	70.01	70.12	69.64
gzip	71.89	68.86	73.62	75.54	69.25	73.55	74.46	74.30	73.47
mcf	82.44	81.30	78.63	72.27	82.13	77.24	70.97	78.40	78.71
parser	64.96	63.23	66.39	68.93	62.72	65.75	66.40	67.62	66.05
twolf	57.78	56.15	58.12	60.20	56.29	57.54	59.52	58.93	58.14
Mean	68.89	67.14	69.28	69.44	67.39	68.71	68.27	**69.87**	69.20

Table 5.2. Prediction accuracies of the modified GAg predictor on unbiased branches.

The following contexts have been used with the modified GAg predictor (Table 5.2):

- GHPC16: the 16 least significant bits of the branch PC (shifted to right by 3 bits) XORed with 16 bits of global history (*gshare* predictor);
- GHLBD16: 16 least significant bits of last branch difference XORed with 16 bits of global branch history;
- LBD4-GHLBD12: 4 least significant bits of last branch difference concatenated with the XOR between 12 least significant bits of last branch difference and 12 bits of global branch history;
- LBD8-GHLBD8: 8 least significant bits of last branch difference concatenated with the XOR between 8 least significant bits of last branch difference and 8 bits of global branch history;
- Shifted-GHLBD16: the 16 least significant bits of last branch difference (shifted to right by 3 bits) XORed with 16 bits of global history;
- Shifted-LBD4-GHLBD12: 4 least significant bits of last branch difference (shifted to right by 3 bits) concatenated with the XOR between 12 least significant bits of last branch difference (shifted to right by 3 bits) and 12 bits of global branch history;
- Shifted-LBD8-GHLBD8: 8 least significant bits of last branch difference (shifted to right by 3 bits) concatenated with the XOR between 8 least significant bits of last branch difference (shifted to right by 3 bits) and 8 bits of global branch history;
- LBD4-GH12: 4 least significant bits of last branch difference concatenated with 12 bits of global branch history;
- Signed-LBD4-GHLBD12: sign bit of last branch difference (0 if positive, 1 if negative) concatenated with 3 least significant bits of last branch difference, and respectively, with the XOR between 12 least significant bits of last branch difference and 12 bits of global branch history.

We have also analyzed a PAg scheme that uses the local (per-address) LBD (last branch difference) by XORing it with the LHR (local history register). The Per-address Branch History Table (PBHT) maintains for each branch its own Local History (LH) and, respectively, its Last Branch Difference (LBD). The predictor is presented in Figure 5.3. Table 4 presents the prediction accuracies obtained with the modified PAg predictor on unbiased branches.

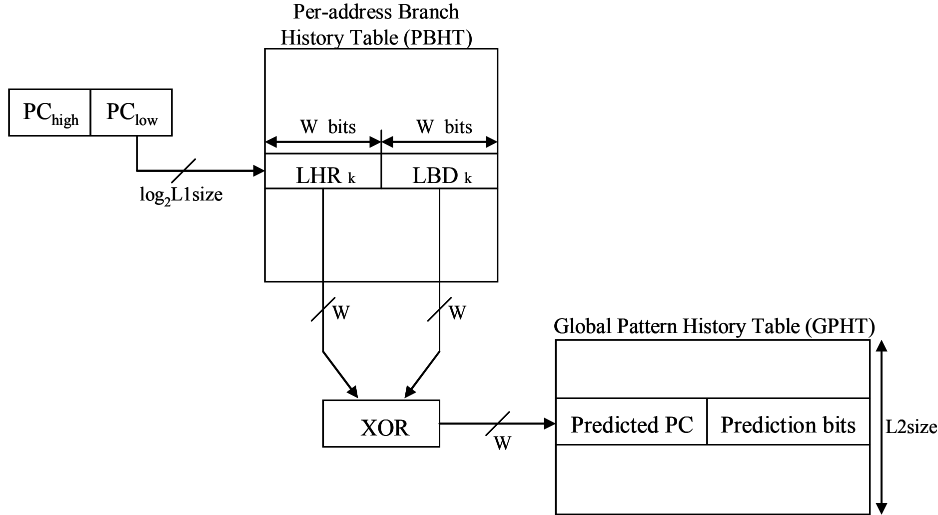

Figure 5.3. The PAg predictor using the local LBD.

Bench	LH16 (PAg)	LHLBD16	LBD4-LHLBD12	LBD8-LHLBD8	Shifted-LHLBD16	Shifted-LBD4-LHLBD12	Shifted-LBD8-LHLBD8	LBD4-LH12	Signed-LBD4-LHLBD12
bzip	74.83	69.86	74.61	74.68	70.07	74.54	74.35	74.80	74.67
gzip	78.37	75.77	79.30	79.62	77.53	78.36	78.48	79.30	79.31
mcf	72.18	70.93	70.55	68.15	73.79	71.91	68.34	69.21	68.76
parser	72.64	74.06	74.82	73.65	72.95	74.30	73.23	73.13	74.52
twolf	68.84	65.75	68.83	69.43	64.60	69.66	70.06	68.16	68.77
Mean	73.37	71.27	73.62	73.11	71.79	**73.75**	72.89	72.92	73.21

Table 5.3. Prediction accuracies of the modified PAg predictor on unbiased branches.

The second level (GPHT) is indexed, depending on the used context, as follows:

- LH16: the second level is indexed by 16 bits of local branch history (PAg predictor);
- LHLBD16: 16 least significant bits of last branch difference XORed with 16 bits of local branch history;
- LBD4-LHLBD12: 4 least significant bits of last branch difference concatenated with the XOR between 12 least significant bits of last branch difference and 12 bits of local branch history;

- LBD8-LHLBD8: 8 least significant bits of last branch difference concatenated with the XOR between 8 least significant bits of last branch difference and 8 bits of local branch history;
- Shifted-LHLBD16: 16 least significant bits of last branch difference (shifted to right by 3 bits) XORed with 16 bits of local history;
- Shifted-LBD4-LHLBD12: 4 least significant bits of last branch difference (shifted to right by 3 bits) concatenated with the XOR between 12 least significant bits of last branch difference (shifted to right by 3 bits) and 12 bits of local branch history;
- Shifted-LBD8-LHLBD8: 8 least significant bits of last branch difference (shifted to right by 3 bits) concatenated with the XOR between 8 least significant bits of last branch difference (shifted to right by 3 bits) and 8 bits of local branch history;
- LBD4-LH12: 4 least significant bits of last branch difference concatenated with 12 bits of local branch history;
- Signed-LBD4-LHLBD12: sign bit of last branch difference (0 if positive, 1 if negative) concatenated with 3 least significant bits of last branch difference, and respectively, with the XOR between 12 least significant bits of last branch difference and 12 bits of local branch history.

Figure 5.4 presents the scheme of the perceptron-based branch predictor that is using as additional prediction information the global last branch difference (LBD). The lower part of the branch address (PC) selects a perceptron in the table of perceptrons and, respectively a local history register in the local branch history table. Thus, local and global branch histories together with the last branch difference are used as inputs for the selected perceptron in order to generate a prediction.

Table 5.4 presents the prediction accuracies obtained with the *piecewise linear branch predictor* on the unbiased branches, using the global LBD as additional prediction information. The global history length is dynamically adjusted between 18 and 48 bits and, respectively, the local history length between 1 and 16 bits, as in [Jim05, Gel07a, Gel07b]. We obtained an unsignificant gain when we used the last branch difference (LBD) entirely (32 bits), even with an increased number of weights from 8590 upto 30713 (the higher weights number being justified by the long additional information).

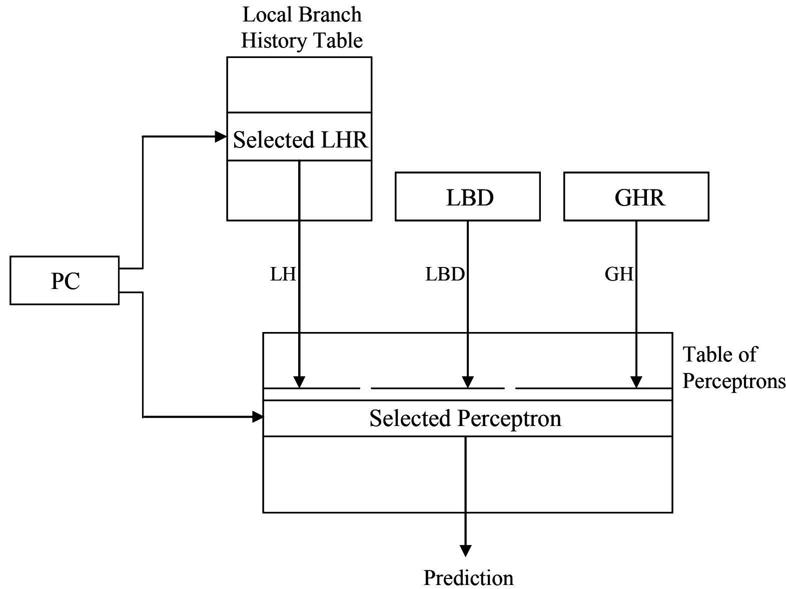

Figure 5.4. Perceptron-based branch predictor using the last known global branch difference.

Bench	GH-LH-8590w	GH-LH-LBD-8590w	GH-LH-LBD-12530w	GH-LH-LBD-15720w	GH-LH-LBD-20573w	GH-LH-LBD-30713w
bzip	76.63%	78.53%	78.58%	78.61%	78.61%	78.64%
gzip	81.29%	81.51%	81.54%	81.54%	81.55%	81.57%
mcf	74.74%	74.79%	74.78%	74.80%	74.79%	74.80%
parser	77.11%	78.31%	78.58%	78.73%	78.84%	78.99%
twolf	76.73%	76.56%	76.77%	77.20%	77.37%	77.52%
Mean	77.30%	77.94%	78.05%	78.18%	78.23%	**78.30%**

Table 5.4. The prediction accuracies obtained with *piecewise linear branch predictor* on unbiased branches, using the global LBD as additional prediction information.

However, with the modified *piecewise linear branch predictor* we obtained a prediction accuracy of 78.30% (see Table 5.4) opposite to those obtained with the modified GAg, 69.87% (see Table 5.2), respectively the modified PAg, 73.75% (see Table 5.3). This gain was probably obtained because both the modified GAg and PAg predictors use a hashing between LBD and global respectively local branch history, while the modified *piecewise linear branch predictor* uses the branch history and LBD without hashing (by concatenating them). Figure 5.5 presents a possible scheme of the perceptron-based branch predictor that is using as prediction information local (per-address) last branch difference (LBD).

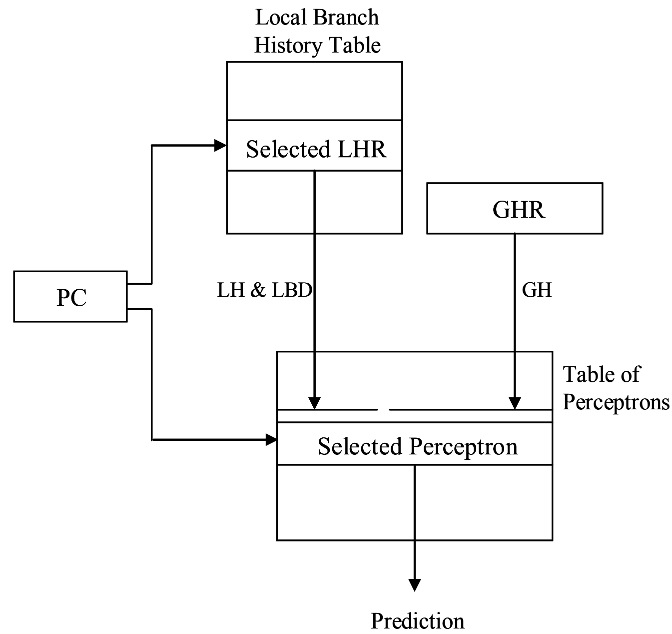

Figure 5.5. Perceptron-based branch predictor using the last known local branch difference.

In Figure 5.5, the Local Branch History Table maintains for each branch its Local History (LH) and, respectively, the Last Branch Difference (LBD). The prediction accuracies obtained with this scheme are presented in Table 5.5.

Bench	**GH-LH-8590w**	**GH-LH-LBD-8590w**	**GH-LH-LBD-12530w**	**GH-LH-LBD-15720w**	**GH-LH-LBD-20573w**	**GH-LH-LBD-30713w**
bzip	76.63%	76.64%	76.67%	76.71%	76.74%	76.77%
gzip	81.29%	81.20%	81.22%	81.23%	81.22%	81.23%
mcf	74.74%	75.00%	74.98%	75.02%	75.00%	75.02%
parser	77.11%	78.00%	78.24%	78.42%	78.56%	78.71%
twolf	76.73%	76.34%	76.53%	76.71%	76.97%	77.24%
Average	77.30%	77.44%	77.53%	77.62%	77.70%	**77.79%**

Table 5.5. Prediction accuracies of the *piecewise linear branch predictor* on unbiased branches, using the local (per-address) LBD as additional prediction information.

Unfortunately, we have not obtained any improvement with the local LBD approach opposite to the global LBD approach, the accuracies being even lower.

6. Designing an Advanced Simulator for Unbiased Branches Prediction

In modern superscalar microarchitectures that speculatively execute a great quantity of code, without performing branch prediction, it won't be possible to aggressively exploit program's instruction level parallelism. Both the architectural and technological complexity of current processors emphasizes the negative impact on performance due to every branch misprediction. Due to this importance, branch prediction becomes a core topic in Computer Architecture curricula. The fast development of computer science and information technology domains, and of computer architecture especially, have determined that many software tools used not far ago in research, to be enhanced with an interactive graphical interface and to be taught in Introductory Computer Organization respectively Computer Architecture courses. The lack of simulators dedicated to branch prediction used in didactical purposes despite of plenty used in research goals, represents the starting point of this paper. The main aim of this section consists in identifying the difficult-to-predict branches, quantifying them at benchmark-level and finding the relevant information to reduce their numbers. Finally, we evaluate the impact of these branches on three commonly used prediction context (local, global and path) and their corresponding predictors ranging from classical two-level predictors to present-day predictors (neural – *Simple Perceptron* and *Fast Path-based Perceptron*). The developed ABPS (Advanced Branch Prediction Simulator) simulator provides a wide variety of configuration options. Beside statistics related to the number of difficult-to-predict branches, the simulator generates graphical results illustrating the influence of different simulation parameters (number of entries in prediction table, history length, etc.) on prediction accuracy, resources usage degree, etc., for every implemented predictor.

Both the architectural complexity of current processors (deep pipeline structures – 20 at INTEL Pentium4 and wide width instruction issue) and technological complexity (higher processing frequency – greater than 3.3 GHz at same processor) emphasize the negative impact on performance due to every branch misprediction [Spr94]. Branch instructions activate at control-flow level generating performance loss by unknowing in

the instruction fetch stage the branch direction and target. Thus, the modern architectures should incorporate very efficacious prediction schemes.

6.1. Simulation Methodology

After more than two decades, the researcher from computer science domain got the conclusion that simulators have become an integral part of the computer architecture research and design process [Yi06]. Their most important advantages, comparing with real processors, are low implementation cost, development time, flexibility and extensibility allowing the architects to quickly evaluate the performance of a wide range of architectures and to quantify the efficacy of every enhancement. Besides its importance proved in computer architecture research field, in the latest time, simulators have been extensively employed as a valuable pedagogical tool as they enable students to visualize how microarchitecture components work and interact [Flo05]. For example, at last important Workshop on Computer Architecture Education held in conjunction with the 33rd International Symposium on Computer Architecture (ISCA06 – the best conference in computer architecture domain in the world), two papers aim at fundamental topics of computer architecture curricula: processor – cache interface in a RISC architecture (MIPS) [Pet06] and power and performance analysis in superscalar out-of-order architecture [Smu06].

In this section we present the implemented ABPS (Advanced Branch Prediction Simulator), an interactive graphical trace-driven simulator for teaching branch prediction [Rad07]. Projects designed around ABPS simulator are used in both undergraduate and graduate level courses at Computer Architecture at “Lucian Blaga” University of Sibiu to teach students concepts related to unbiased branch, state of the art branch predictors, branch prediction constraints and limits of instruction level parallelism. Our approach in teaching branch prediction represents a formative necessity since computer architecture is mainly approached in a descriptive manner. Through our approach students have the opportunities to be creative / innovative in computer architecture or in other fundamental research / didactical domains of computer science and information technology, even in countries not very developed from economical point of view. Based on highly parameterized developed simulation tools, students can understand more in depth the theoretical concepts related to branch prediction constraints, limits of instruction level parallelism. It could be observed the different benchmarks’ influence on every proposed architectural innovation.

Unfortunately, this version of the simulator uses only an analytical model to determine the impact of unbiased branch and branch missprediction on global processing performance [Vin07]. In his model, related to a superscalar processor, Vintan ignores stalls like cache misses and bus conflicts focalizing only about the penalty introduced by branch miss-prediction. In their assignments, students are asked to explore architecture configurations extending them for optimizing the power, performance, or both within a given chip area budget (based on other simulation tools – CACTI, WATTCH [Shi01, Bro00]).The simulator code is open source and can be found at [ABPS07].

The simulator allows trace-driven simulation on a collection of 17 programs (having 1 million of dynamic branch instructions each) from different versions of SPEC benchmarks [SPEC]. We use all of the SPEC CPU2000 integer benchmarks, and all of the SPEC CPU95 integer benchmarks that are not duplicated in SPEC CPU2000. The benchmarks are compiled with the *CompaQ GEM* compiler with the optimization flags *-fast -O4 -arch ev6* [Coh00]. All these benchmarks cover a lot of applications ranging from compression (text/image) to word processing, from compilers and architectures to games enhanced with artificial intelligence, etc.

From a pedagogical point of view, the proposed tool benefits the learning process because it helps students to observe the influence of each parameter on simulation model. The simulator provides a wider variety of configuration options. Thus, it can be determined how the prediction accuracy does vary with input parameters (number of entries in prediction tables, history length, number of bits for weights representation, threshold value used for perceptron training, etc). The ABPS simulator assures three of the features specific to almost high-performance standard simulators: free availability for use, extensibility and portability. Full inheritance and polymorphism is used, allowing for ease of extension in the future adding new functionalities.

6.2. The Functional Kernel of the Simulator

The realized simulator must remove the bottlenecks that limit the processor performance and search for possible changes (architectural or optimization techniques) for improving it. Providing a highly parameterized model for every microarchitectural instance, the performance obtained by simulation will represent a quick feedback mechanism related to proposed changes. The simulator execution consists in the following sequential steps:

1) *Configuring the microarchitecture with the input parameters including the benchmarks*.
2) *Initialization phase* (prediction tables, local/global history registers).
3) *Starting the trace processing and computing the simulation metrics.*

The mechanism that identifies unbiased branches was already presented in Chapter 3. The *Detector* kernel of ABPS finds the unbiased branches (those that have their polarization index – the percentage of “not taken” or “taken” branch instances corresponding to a certain context – lower than a *polarization degree*, set prior the simulation) and quantifies their number. Repeating the unbiased branches detection methodology for a length-ordered set of contexts it could be observed how the number of unbiased branches decreases.

The prediction process supposes accessing the tables for every instruction from traces and establishing the prediction function of associated prediction automaton or perceptron. Every good prediction does increase the automatons state or perceptron weights, while every misprediction does decrease the same parameters. The automatons are implemented as saturating counters and, in the neural predictors’ case, the threshold keeps from overtraining, permitting the perceptron to adapt quickly at every changing behavior.

6.3. The Software Design of the ABPS Simulator

The user diagram (Figure 6.1a) illustrates the general user interaction process with ABPS. A generic user can mainly interact with ABPS in two ways (not fully distinct):

- *Default start* – the user starts a simulation using the default input parameters.
- *Custom start (Choose simulation type)* – the user chooses:

  1. The simulation type – detection or prediction;
  2. The benchmarks (Stanford and/or SPEC 2000);
  3. The values for the simulation parameters.

Steps 1, 2, 3 can be executed in any order. Either of steps 1 and 3 is not mandatory. If one of them is not executed, default values are used. Step 2 (choosing the benchmarks) is necessary the first time (initially no traces are selected for simulation) for both user interaction types. After the three steps

removed

- *Starts simulation* – the simulation begins after all the inputs had been set. The simulation process consists basically in processing each trace included (in a multithreaded manner);
- *Read trace* – each trace is processed, branch after branch. Each branch instruction is fed to the selected detector / predictor. This is done until all branch instructions (from the selected trace) are processed. During this, results are accumulated.
- *Processing results* – after a trace had been processed, the obtained results are processed in order to compute certain metrics;
- *Display results* – the results are displayed and the simulation process stops.

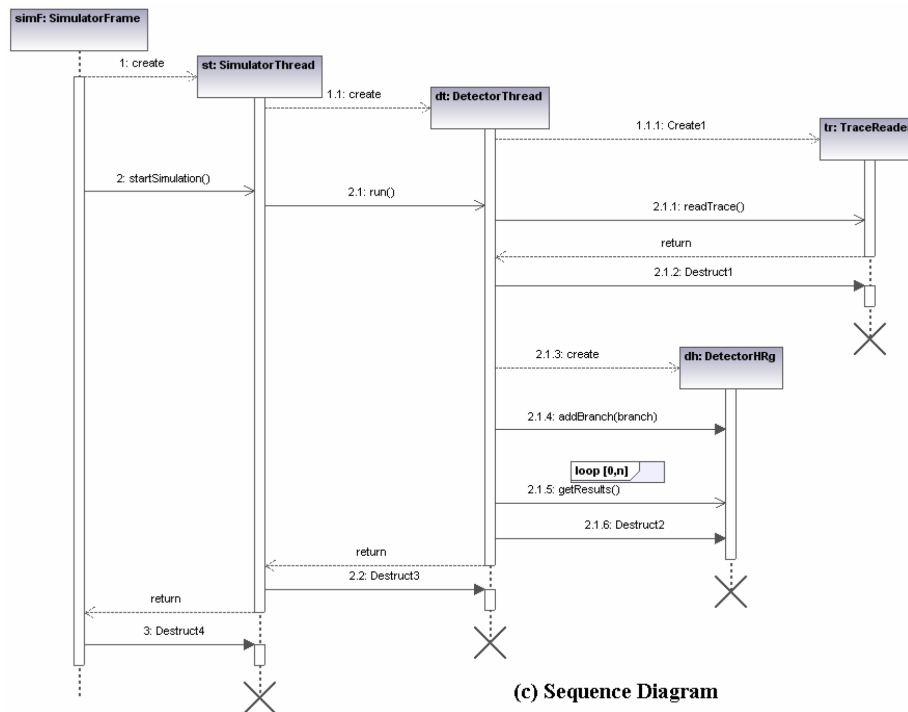

Figure 6.2. Sequence Diagram.

The sequence diagram (Figure 6.2) presents in detail how ABPS performs the process of detecting unbiased branches. The process starts in the GUI, where the detection parameters are set. After this initialization, the user can trigger the detection process, which will be managed by another thread (*1:*

create, st:SimulatorThread). In this way, the GUI will not block itself, leaving the user with the ability to perform other tasks from ABPS. The simulation thread will create and start a detection thread (1.1: create, dt:DetectorThread). The detection thread will manage all the detection process (1.1.1: Create1, tr:TraceReader). When all the above initializations were performed, the detection process actually starts (2: startSimulation(), 2.1: run()): the trace used for simulation is processed using the appropriate detector (see: 2.1.1 – 2.1.6). Finally, the detection thread signals (by returning the results) the simulation thread that the detection is done (2.2: Destruct3). In the same manner, the simulation thread signals the GUI thread (3:Destruct4), which will display the results.

From the user's point of view it is very necessary a visual friendly interface, based on menus, butons, dialog boxes, graphical images. The simulator must be easy to use and the results must be efficiently interpreted and processed (eventually transferred to some utility application such Excel, PowerPoint, Internet). The machine model should be "*fine-tuned*" to remove redundant or little hardware features and to investigate possible tradeoffs of performance against the functionality provided.

To run the ABPS simulator, on the host computer the *jre-1_5* (or higher) or *jdk-1_5* (or higher) must first installed. ABPS is written in JAVA, thus is platform independent. For properly use of ABPS simulator it should be accomplished some system requirements. Thus, it is recommended to have a processor with at least 1 GHz frequency. Otherwise, due to JVM (java virtual machine), the simulation time, especially on SPEC2000 benchmarks, risk to become prohibit. The RAM memory recommended is 256Mbytes. Since we can represent on the same chart up to 17 benchmarks (even 6 bars on each), to have a good view it is required a 1024x768 minimum screen resolution.

The ABPS simulator is organized around a main window that contains two panels. The left one is used to configure (initialize all requested parameters) and control simulation. The right panel is based on two tabs – one that show every simulations' results in text format, and another, that permits to generate graphical charts illustrating the influence of different simulation parameters on metrics like unbiased branches percentage, prediction accuracy, processing rate. The left panel is divided in two parts: the upper part contains the available testing programs. The *Remove* respectively *Add* buttons facilitate to remove the selected benchmark or to add new ones. The user can opt to choose between Stanford or SPEC benchmarks, single or multiple selections. Any simulation started will operate exclusively on selected benchmarks. Also, there are two very expressive buttons that allow *select*ing or *deselect*ing all benchmarks. The

lower part of left panel contains two tabs *Detector / Predictor*, each having its own configuring parameters. The inputs for *Detector* are: the *global history length* – GH, the *local history length* – LH, a flag that show if path information correlation is used (concatenated), and the *polarization degree* of each context instance. The *Predictor* tab contains its own four tabs specific to each implemented predictor (*GAg, PAg, PAp* and *Perceptron*). The implemented two-level predictors request as inputs parameters: the number of entries in prediction table, the history length (global / local). Besides input parameters used by the two-level predictors, the neural predictors (*Simple Perceptron* and *Fast Path-based Perceptron*) need some additionals: threshold value used for learning algorithm, number of bits for storing the weights. Each predictor can predict all branches or only unbiased branches. If the second choice is made the simulator apply first the *Detector* phase, hidden for user. After determining the unbiased branches percentage the performance loss can be computed comparatively with an equivalent multiple instruction issue processor having an ideal branch predictor.

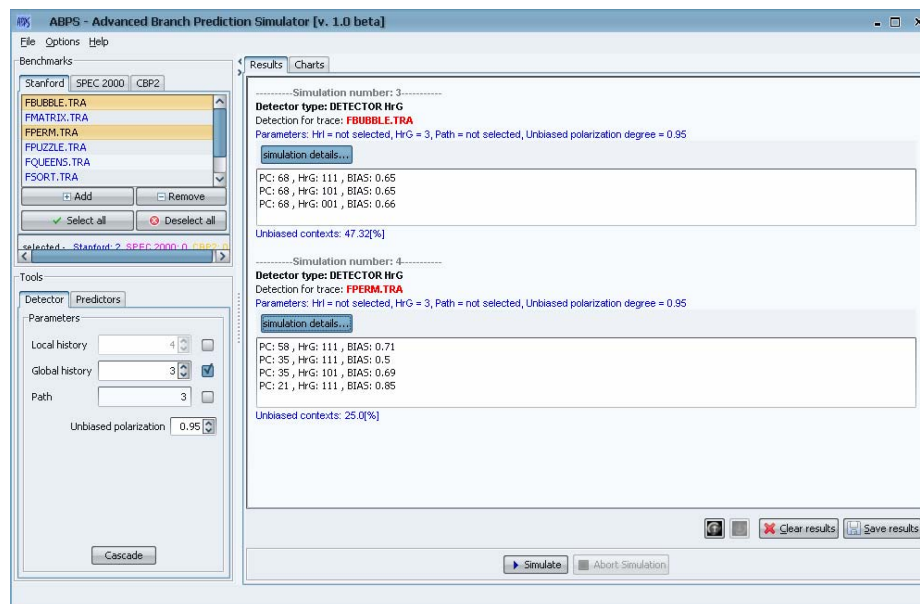

Figure 6.3. ABPS simulator – unbiased branches detection.

If the user chooses from Configuration panel the *Detector* tab and in the Results panel only simple execution (*Simulate* buton), among the simulation results a list of unbiased branches, in their certain contexts, does occure. This list could be saved (in *text* or *csv* format) for further analysis between

different unbiased branches lists obtained when the contexts length is extended. An important result is the unbiased branches percentage from the tested benchmarks. The students can see how this percentage does vary when the context length changes. Figure 6.3 shows the simulation results when the Detector tab was selected.

Figure 6.4. ABPS simulator – variation of prediction accuracy with global history length.

If the user selected from Configuration panel the *Predictors / Perceptron* tab (*Simple* or *Fast Path-based*) and in the Results panel only simple execution (not charts generating), the simulation results consist in four important metrics. The prediction accuracy is the number of correct predictions divided to total number of dynamic branches. We compute also a confidence metric that represents the total cases when the prediction was correct and the perceptron did not need to be trained (the magnitude of the perceptron output was greater than the threshold), divided to total number of correct predictions (therefore, considering a trivial threshold equal with 0). While the first two have impact on processor's performance, the next two metrics have direct influence on transistors' budget and integration area (the number of perceptrons used in the prediction process and respectively the saturation degree of the perceptrons). The saturation degree represents the percentage of cases when the weights of perceptrons cannot be increased /

decreased because they are saturated. If the last two metrics are quite low, it means that the perceptrons are underused. The prediction accuracy and the usage degree of prediction table are also computed in the case of two-level predictors.

The *Charts* tab offers the possibility to illustrate graphical simulation results. From the two listboxes the user can select which metrics (from those explained earlier) to be used and which input parameter to be varied on all selected benchmarks. An interesting chart shows the Issue Rate (IR) relative speedup obtained by growing the context length. We used the formula [IR(L)–IR(16)]/IR(16), for computing IR relative speedup, where L is the context's length, $L \in \{20, 24, 28, 32\}$). The last group of columns represents the average (or geometric / harmonic mean). The chart type may be *Bar* or *Line*. The chart can be saved in *png* format just by clicking on *SaveChart* button. Figure 6.4 illustrates how the prediction accuracy does vary with the global history length when the *Fast Path-based Perceptron* predictor is used on all Stanford benchmarks.

7. Conclusions and Further Work

Based on laborious simulations we showed that the percentages of difficult branches are quite significant (at average between 6% and 24%, depending on the different used contexts and their lengths). The simulations also show that the path is relevant for better polarization rate and prediction accuracy only in the case of short contexts. As Figures 3.5 and 3.6 suggest, our conclusion is that despite some branches are path-correlated, long history contexts (local and global) approximate well the path information. In other words, sufficient long history contexts might be viewed as a good "compression" of the most complete path information. In our further work, we'll try to reduce the path information extracting and using only the most important bits. Thus, the path information could be built using only a part of the branch address instead of all the 32 bits of the complete PC.

Therefore, it is obvious that for the unbiased branches identified in [Gel06, Vin06] the prediction information used by the present-day branch predictors (local/global correlations and path information), is not always sufficiently relevant and, therefore, these branches cannot be accurately predicted. Using the perceptron predictor we measured on these unbiased branches an average prediction accuracy of only 73.46% (and 92.58% on all branches). We also evaluated the Frankenpredictor [Loh05a], the O-GEHL [Sez05], and the Piecewise Linear Branch Predictor [Jim05] on unbiased branches, but the prediction accuracy was still low despite these predictors are using path-based information too. Using the Piecewise Linear Branch Predictor we obtained a prediction accuracy of 77.30% on the unbiased branches (94.92% on all branches) from the SPEC2000 benchmarks. Therefore, we introduced new prediction information, named branch difference history, representing the history of branch conditions' signs. Our first goal was to exploit the correlation existing between the history of conditions' signs (negative, zero or positive) encountered by a certain branch instruction and the next condition's sign corresponding to that branch. If the condition sign is predictable, the branch's behavior is predictable too because branch's output is deterministically correlated with the condition's sign.

Thus, we implemented a local branch difference predictor using the *Prediction by Partial Matching* (PPM) algorithm. We determined through simulations that the optimal configuration of the predictor consists in a

Branch Difference History Table with 256 entries, a history length of 24 values, and a pattern length of 3. We obtained with this scheme on the unbiased branches an average branch difference prediction accuracy of 68.60% and a final branch prediction accuracy of 71.76% (90.55% on all branches). However, when we used a threshold of 7, we obtained a final branch prediction accuracy of 79.69% on unbiased branches (and 96.43% on all branches). Our combined global and local approach associates to each global difference history pattern its own BDHT. Evaluating this scheme on unbiased branches, we obtained a final branch prediction accuracy of 71.54% (92.33% on all branches) without threshold, and, respectively, 81.25% (97.44% on all branches) with a threshold of 7. Finally, with the branch difference prediction scheme that combines multiple partial matches, we obtained a final branch prediction accuracy of 72.24% on the unbiased branches (and 91.59% on all branches), without threshold.

Further we show that the last branch condition is more efficient than the path information: it decreased the percentage of unbiased branches for all evaluated context lengths. Therefore we used this new prediction information in some state-of-the-art branch predictors. Unfortunately, the improvement obtained using the LBD entirely (32 bits), in terms of prediction accuracy, is not significant.

Finally, we presented our ABPS simulator. Repeating the detection methodology for a length-ordered set of contexts it could be observed how the number of unbiased branches decreased, in the tested benchmarks. Another facility of ABPS consists in running a plenty of branch predictors, from classical two-level up to neural state-of-the art, having the possibility of varying the most important parameters and illustrating the graphical results of the simulations. Also important, our simulator permits the migration of some mature actual scientific problems to students' understanding level.

In conclusion the average prediction accuracy remains still low on unbiased branches. During this work, we showed that difficult branches were efficiently identified in [Gel06, Vin06]. Furthermore, the accurate prediction of these unbiased branches constitutes an open problem, since each percent of unbiased branches decisively reduces prediction accuracy. As a consequence, these unbiased branches might define a fundamental limit in branch prediction research.

Further Work

We consider that the use of more prediction contexts (some HLL code information) is required to further improve prediction accuracies. In order to efficiently use such information we consider it will be necessary to have a significant amount of compiler support. Another alternative could be to pursue the concepts of micro-threading where small fragments of code are executed concurrently and the branch problem is no longer a major concern. Also, we want to explore the importance of unbiased branch prediction problem in Chip Multi-Processor (CMP) architectures.

For further work we are concerned to the necessity of an efficient hardware branch predictor from power consumption and performance criterions, within a given chip area budget. Very high prediction accuracy is necessary, because taking into account the multiple-instruction-issue processors characteristics as pipeline depth or issue rates, even a prediction miss rate of a few percent involves a substantial performance loss. Also, we intend to extend the ABPS simulator with functional network characteristics, allowing a distributed simulation process in a client-server manner, useful due to the time consuming simulations.

Another objective is to develop a complex architecture that selectively anticipates the values produced by high-latency instructions. We will focalize on multiply, division and loads that access with miss the L2 data cache. The DIV/MUL instructions (non-selective approach) will be solved by an Instruction Reuse scheme, without prediction. The critical load instructions (loads with miss in both cache levels – selective approach) will be solved by a reuse scheme or, if they are not reusable, through prediction (a simple prediction scheme will be used, e.g. last value predictor). We will evaluate this complex architecture and compare it with a *blocked multithreading* architecture.

References

[Aam03] Aamer M., Lux K., Mistry R., Mulholland B., *Efficiency of Pre-Computed Branches*, Technical Report, University of Pennsylvania, USA, 2003.

[ABPS07] Advanced Branch Prediction Simulator, http://webspace.ulbsibiu.ro/adrian.florea/html/simulatoare/simulatoare.htm.

[Ara01] Aragón J. L., González J., García J. M., González A., *Selective Branch Prediction Reversal by Correlating with Data Values and Control Flow*, Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors, 2001.

[Bro00] Brooks D., Tiwari V., Martonosi M., *Wattch: a framework for architectural-level power analysis and optimizations*. In Annual International Symposium on Computer Architecture, pages 83–94, 2000.

[CBP04] *The First Championship Branch Prediction Competition (CBP-1)*, http://www.jilp.org/cbp, 2004.

[CBP06] *The* Second *Championship Branch Prediction Competition (CBP-2)*, http://www.jilp.org/cbp, 2006.

[Cha94] Chang P.-Y., Hao E., Yeh T.-Y., Patt Y. N., *Branch Classification: a New Mechanism for Improving Branch Predictor Performance*, Proceedings of the 27th International Symposium on Microarchitecture, San Jose, California, 1994.

[Cha02] Chappell R., Tseng F., Yoaz A., Patt Y., *Difficult-Path Branch Prediction Using Subordinate Microthreads*, The 29th Annual International Symposia on Computer Architecture, Alaska, USA, May 2002.

[Cha03] Chaver D., Pinuel L., Prieto M., Tirado F., Huang M., *Branch Prediction On Demand: an Energy-Efficient Solution*, ISLPED'03 Conference, Seoul, Korea, August 2003.

[Che03] Chen L., Dropsho S., Albonesi D.H., *Dynamic Data Dependence Tracking and its Application to Branch Prediction*, The 9th International Symposium on High-Performance Computer Architecture, February 2003.

[Coh00] Cohn R., Lowney P. G., *Design and Analysis of Profile-Based Optimization in Compaq's Compilation Tools for Alpha*, Journal of Instruction-Level Parallelism nr 3, 2000.

[Con04] Constantinides K., Sazeides Y., *A Hardware-Based Method for Dynamically Detecting Instruction-Isomorphism and its Application to Branch Prediction*, The 2nd Value Prediction and Value-Based Optimization Workshop, Boston, Massachusetts, October 2004.

[Des04] Desmet V., Eeckhout L., De Bosschere K., *Evaluation of the Gini-index for Studying Branch Prediction Features.* Proceedings of the 6th International Conference on Computing Anticipatory Systems (CASYS), AIP Conference Proceedings, Vol. 718, 2004.

[Des06] Desmet V., *On the Systematic Design of Cost-Effective Branch Prediction*, PhD Thesis, Ghent University, Belgium, 2006.

[Ega03] Egan C., Steven G., Quick P., Anguera R., Vintan L., *Two-Level Branch Prediction using Neural Networks*, Journal of Systems Architecture, vol. 49, issues 12-15, Elsevier, December 2003.

[Flo05] Florea A., *The dynamic values prediction in the next generation microprocessors*, MatrixRom Publishing House, Bucharest, 2005.

[Flo07] Florea A., Radu C., Calborean H., Crapciu A., Gellert A., Vintan L., *Designing an Advanced Simulator for Unbiased Branches' Prediction*, The 9th International Symposium on Automatic Control and Computer Science, Iasi, November 2007.

[Gao06] Gao H., Zhou H., *PMPM: Prediction by Combining Multiple Partial Matches*, The 2nd Journal of Instruction-Level Parallelism Championship Branch Prediction Competition (CBP-2), Orlando, Florida, USA, December 2006.

[Gel06] Gellert A., *Prediction Methods Integrated into Advanced Architectures*, Technical Report, Computer Science Department, "Lucian Blaga" University of Sibiu, January 2006.

[Gel07a] Gellert A., *Integration of Some Advanced Prediction Methods into Speculative Computing Systems*, Technical Report, Computer Science Department, "Lucian Blaga" University of Sibiu, March 2007.

[Gel07b] Gellert A., Florea A., Vintan M., Egan C., Vintan L., *Unbiased Branches: An Open Problem*, Twelfth Asia-Pacific Computer Systems Architecture Conference (ACSAC'07), Seoul, Korea, August 2007.

[Gon99] González J., González A., *Control-Flow Speculation through Value Prediction for Superscalar Processors*, International Conference on Parallel Architecture and Compilation Techniques, 1999.

[Gon01] González J., González A., *Control-Flow Speculation through Value Prediction*, IEEE Transactions on Computers, Vol. 50, No. 12, December 2001.

[Hei99a] Heil T., Smith Z., Smith J.E., *Using Data Values to Predict Branches*, Proceedings of the 26th Annual International Symposium on Computer Architecture, 1999.

[Hei99b] Heil T.H., Smith Z., Smith J.E., *Improving Branch Predictors by Correlating on Data Values*, The 32nd International Symposium on Microarchitecture, November 1999.

[Jim01] Jiménez D., Lin C., *Dynamic Branch Prediction with Perceptrons*, In Proceedings of the Seventh International Symposium on High Performance Computer Architecture (HPCA-7), January 2001.

[Jim02] Jiménez D., Lin C., *Neural Methods for Dynamic Branch Prediction*, ACM Transactions on Computer Systems, Vol. 20, New York, USA, November 2002.

[Jim03] Jiménez D., *Fast Path-Based Neural Branch Prediction*, Proceedings of the 36th Annual International Symposium on Microarchitecture, December 2003.

[Jim05] Jiménez D., *Idealized Piecewise Linear Branch Prediction*, Journal of Instruction-Level Parallelism, April 2005.

[Loh05a] Loh G. H., *Deconstructing the Frankenpredictor for Implementable Branch Predictors*, Journal of Instruction-Level Parallelism, April 2005.

[Loh05b] Loh G. H., Jiménez D., *A Simple Divide-and-Conquer Approach for Neural-Class Branch Prediction*, Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT), St. Louis, MO, USA, September 2005.

[Loh05c] Loh G. H., Jiménez D., *Reducing the Power and Complexity of Path-Based Neural Branch Prediction*, 5th Workshop on Complexity Effective Design (WCED5), Madison, WI, USA, June 2005.

[Mah94] Mahlke S. A., Hank R. E., Bringmann R. A., Gyllenhaal J. C., Gallagher D. M., Hwu W.-M. W., *Characterizing the Impact of Predicated Execution on Branch Prediction*, Proceedings of the 27th International Symposium on Microarchitecture, San Jose, California, December 1994.

[McFar93] McFarling S., *Combining Branch Predictors*, WRL Technical Note TN-36, Digital Equipment Corporation, June 1993.

[Mud96] Mudge T. N., Chen I. K., Coffey J. T., *Limits to Branch Prediction*, Technical Report, Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, Michigan, USA, January 1996.

[Nair95] Nair R., *Dynamic Path-Based Branch Correlation*, IEEE Proceedings of MICRO-28, 1995.

[Oan06] Oancea M., Gellert A., Florea A., Vintan L., *Analyzing Branch Prediction Contexts Influence*, Advanced Computer Architecture and Compilation for Embedded Systems, (ACACES 2006), ISBN 90 382 0981 9, pages 5-8, L'Aquila, Italy, July 2006.

[Pet06] Petit S., Tomás N., Sahuquillo J., Pont A., *An Execution-Driven Simulation Tool for Teaching Cache Memories in Introductory Computer Organization Courses*, Workshop on Computer Architecture Education, Boston, 2006.

[Rab89] Rabiner L. R., *A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition*, Proceedings of the IEEE, Vol 77, No. 2, February 1989.

[Rad07] Radu C., Calborean H., Crapciu A., Gellert A., Florea A., *An Interactive Graphical Trace-Driven Simulator for Teaching Branch Prediction in Computer Architecture*, The 6th EUROSIM Congress on Modelling and Simulation, Ljubljana, Slovenia, September 2007.

[Rot99] Roth A., Moshovos A., Sohi G., *Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation*, Proceedings of International Conference on Supercomputing, 1999.

[Saz97] Sazeides Y., Smith J. E., *The Predictability of Data* Values, Proceedings of the 30th Annual International Symposium on Microarchitecture, December 1997.

[Sez02] Seznec A., Felix S., Krishnan V., Sazeides Y., *Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor*, Proceedings of the 29th International Symposium on Computer Architecture, Anchorage, AK, USA, May 2002.

[Sez05] Seznec A., *Genesis of the O-GEHL branch predictor*, Journal of Instruction-Level Parallelism, April 2005.

[Shi01] Shivakumar P., Jouppi N. P., *CACTI 3.0: An Integrated Cache Timing, Power, and Area Model*, WRL Technical Report 2001/2.

[Sim] Simplescalar, *The SimpleSim Tool Set*, ftp://ftp.cs.wisc.edu/pub/sohi/Code/simplescalar.

[Sin06] Singer J., Brown G., *Return Value Prediction Meets Information Theory*, The 4th Workshop on Quantitative Aspects of Programming Languages, Vienna, Austria, April 2006.

[Smi98] Smith Z., *Using Data Values to Aid Branch-Prediction*, MSc Thesis, Wisconsin-Madison, USA, December 1998.

[Smu06] Smullen C.W., Taha T.M., *PSATSim: An Interactive Graphical Superscalar Architecture Simulator for Power and Performance Analysis*, Workshop on Computer Architecture Education, Boston, 2006.

[SPEC] SPEC2000, *The SPEC benchmark programs*, http://www.spec.org.

[Spr94] Sprangle E., Carmean D., *Increasing processor performance by implementing deeper pipelines*, 29th International Symposium on Computer Architecture, Anchorage, Alaska, May 25 - 29, 2002.

[Sri06] Srinivasan R., Frachtenberg E., Lubeck O., Pakin S., Cook J., *Neuro-PPM Branch Prediction*, The 2nd Journal of Instruction-Level Parallelism Championship Branch Prediction Competition (CBP-2), Orlando, Florida, USA, December 2006.

[Ste97] Steven G.B., Christian B., Collins R., Potter R.D., Steven F.L., *A Superscalar Architecture to Exploit Instruction Level Parallelism*, Microprocessors and Microsystems, 1997.

[Tar05] Tarjan D., Skadron K., *Merging Path and GshareIndexing in Perceptron Branch Prediction*, ACM Transactions on Architecture and Code Optimization, Vol. 2, No. 3, September 2005.

[Tho03] Thomas R., Franklin M., Wilkerson C., Stark J., *Improving Branch Prediction by Dynamic Dataflow-based Identification of Correlated Branches from a Large Global History*, Proceedings of the 30th International Symposium on Computer Architecture, June 2003.

[Tho01] Thomas R., Franklin M., *Using Dataflow Based Context for Accurate Value Prediction*, Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2001.

[Vin99a] Vintan L., Iridon M., *Towards a High Performance Neural Branch Predictor*, International Joint Conference on Neural Networks, Washington DC, USA, July 1999.

[Vin99b] Vintan L., Egan C., *Extending Correlation in Branch Prediction Schemes*, International Euromicro'99 Conference, Italy, September 1999.

[Vin03] Vintan L., Sbera M., Mihu I.Z., Florea A., *An Alternative to Branch Prediction: Pre-Computed Branches*, ACM SIGARCH Computer Architecture News, Vol.31, Issue 3, ACM Press, NY, USA, June 2003.

[Vin05] Vintan L., Florea A., Gellert A, *Focalising Dynamic Value Prediction to CPU's Context*, IEE Proceedings. Computers & Digital Techniques, Vol. 152, No. 4, Stevenage, UK, July 2005.

[Vin06] Vintan L., Gellert A., Florea A., Oancea M., Egan C., *Understanding Prediction Limits through Unbiased Branches*, Eleventh Asia-Pacific Computer Systems Architecture Conference (ACSAC'06), Shanghai, China, September 2006.

[Vin07] Vintan L., *Prediction Techniques in Advanced Computing Architectures* (in English), MatrixRom Publishing House, Bucharest, 2007.

[Wan97] Wang K., Franklin M., *Highly Accurate Data Value Prediction using Hybrid Predictors*, Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, December 1997.

[Yeh92] Yeh T.-Y., Patt Y. N., *Alternative Implementations of Two-Level Adaptive Branch Prediction*, Proceedings of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia, May 1992.

[Yi06] Yi J.J., Lilja D.J., *Simulation of Computer Architectures: Simulators, Benchmarks, Methodologies and Recommendations*, IEEE Transactions on Computers, Vol. 55, No. 3, March 2006, pp. 268-280.

Glossary

Benchmark: is a program used for evaluations. In this work we used the SPEC2000 benchmark suite and the CBP-1 traces.

Biased branch: mostly always taken or mostly always not taken branch (mostly-one-direction branch). The behavior (taken/not taken) of a biased branch is polarized.

Biased branch context: the branch behavior (taken/not taken) is polarized for that certain context (local branch history, global history, etc.).

Blocked multithreading: a multithreading architecture which switches threads at high latency instructions (e.g. *critical loads*).

Branch difference: represents the value or the sign of the difference between the branch's inputs. Regarding the sign of the inputs' difference, a value of 1 indicates that the corresponding branch difference is positive, a value of -1 indicates a negative difference, while a 0 indicates equality between the branch's inputs.

Branch difference predictor: the branch outcomes are predicted based on *branch difference* histories.

Branch polarization: measured through the *polarization index* (P).

Branch prediction: is the prediction of the direction (taken/not taken) and/or the target address (next PC) of a branch instruction.

Complete-PPM predictor: see *Prediction by Partial Matching (PPM).*

Confidence automaton: saturated counter that indicates the confidence of a certain prediction. The prediction is generated only if the confidence automaton is in a predictable state.

Context: the context of length *p* represents the last *p* elements from the correlation information used in order to make a prediction. In the case of person movement prediction the correlation information is the room history, and a context of length *p* consists in the last *p* visited rooms. In the case of branch prediction the correlation information is the branch history (e.g. local or global branch history), and a context of length *p* consists in the last *p* bits from the branch history.

Context instance: is a dynamic branch executed in the respective context.

Critical load: a load instruction with miss in both cache levels.

Distribution (index): the distribution index of a certain branch context is computed as follows.

$$D(S_i) = \begin{cases} 0, & n_t = 0 \\ \dfrac{n_t}{2 \cdot \min(NT, T)}, & n_t > 0 \end{cases}$$, where

- n_t = the number of branch outcome transitions, from taken to not taken and vice-versa, in context S_i;
- $2 \cdot \min(NT, T)$ = maximum number of possible transitions;
- k = number of distinct contexts, $k \le 2^p$, where p is the length of the binary context;
- if $D(S_i) = 1, (\forall) i = 1, 2, ..., k$, then the behavior of the branch in context S_i is "contradictory" (the most unfavorable case), and thus its learning is impossible;
- if $D(S_i) = 0, (\forall) i = 1, 2, ..., k$, then the behavior of the branch in context S_i is constant (the most favorable case), and it can be learned.

Dynamic branch: is an instance of a *static branch* during program execution.

Dynamic branch prediction: the branches are predicted with hardware techniques.

Dynamic learning: is the run-time prediction process when the outputs of the predictor are used to adjust the prediction structures and respectively to generate predictions.

Feature (set): is the binary context on p bits of prediction information such as local history, global history or path. Each static branch finally has associated k dynamic contexts in which it can appear ($k \le 2^p$).

Gain: is the factor which gives the improvement of the quality.

Last branch difference (LBD): a branch condition difference consists in the difference of the operand values implied in the last branch condition. The global LBD is the last known branch condition difference. The local LBD is the last per-address branch condition difference.

Markov chain: in the case of a first order Markov chain the probabilistic description is truncated to just the current and predecessor state. $P[q_t = S_j | q_{t-1} = S_i,\ q_{t-2} = S_k,\ ...] = P[q_t = S_j | q_{t-1} = S_i]$, where q_t is the state at time t. Thus, for a first order Markov chain with N states, the set of transition probabilities between states S_i and S_j is $A = \{a_{ij}\}$, where $a_{ij} = P[q_t = S_j | q_{t-1} = S_i]$, $1 \le i,\ j \le N$, having the properties $a_{ij} \ge 0$

and $\sum_{j=1}^{N} a_{ij} = 1$. For a Markov chain of order R the probabilistic description is truncated to the current and R previous states.

Markov predictor: the prediction is generated based on the state transition probabilities of a *Markov chain*.

Polarization (index): the polarization index (P) of a certain branch context is computed as follows.

$P(S_i) = \max(f_0, f_1) = \begin{cases} f_0, & f_0 \geq 0.5 \\ f_1, & f_0 < 0.5 \end{cases}$, where

- $S = \{S_1, S_2, ..., S_k\}$ = set of distinct contexts that appear during all branch instances;
- k = number of distinct contexts, $k \leq 2^p$, where p is the length of the binary context;
- $f_0 = \frac{T}{T+NT}$, $f_1 = \frac{NT}{T+NT}$, NT = number of "not taken" branch instances corresponding to context S_i, T = number of "taken" branch instances corresponding to context S_i, $(\forall) i = 1, 2, ..., k$, and obviously $f_0 + f_1 = 1$;
- if $P(S_i) = 1, (\forall) i = 1, 2, ..., k$, then the context S_i is completely biased (100%), and thus, the afferent branch is highly predictable;
- if $P(S_i) = 0.5, (\forall) i = 1, 2, ..., k$, then the context S_i is totally unbiased, and thus, the afferent branch is not predictable if the taken and not taken outcomes are shuffled.

Prediction accuracy: the percentage or ratio of correct predictions reported to the total number of predictions.

Prediction by Partial Matching (PPM): is a context-based prediction algorithm. The PPM predictor contains a set of simple Markov predictors. It is predicted the value that followed the context with the highest frequency. In the case of complete-PPM predictor, if a prediction cannot be generated with the Markov predictor of order *k*, then the pattern length is shortened and the Markov predictor of order *k-1* tries to predict and so on.

Speculative execution: instruction execution based on predicted values or predicted branch outcomes.

Static branch: a certain branch instruction from a program.

Static branch prediction: the branches are predicted statically by the compiler. Static branch predictors are used in processors where the expectation is that branch behavior is highly predictable at compile-time.

Static learning: means that before effective run-time prediction process, the predictor is trained based on some patterns. In the static learning process the outputs of the predictor are used only to adjust the prediction structures.

Unbiased branch: a branch whose behavior (taken/not taken) is not sufficiently polarized.

Unbiased branch context: the branch behavior (taken/not taken) is not sufficiently polarized for that certain context (local branch history, global history, etc.).